pake Documentation

Release 0.17.0.3a1

Teriks

Aug 23, 2019

Contents

pake is a make-like python build utility where tasks, dependencies and build commands can be expressed entirely in python, similar to ruby rake.

pake supports automatic file/directory change detection when dealing with task inputs and outputs, and also parallel builds.

pake requires python3.5+

CHAPTER 1

Installing

Note: pake is Alpha and likely to change some.

To install the latest release use:

```
sudo pip3 install python-pake --upgrade
```

If you want to install the development branch you can use:

```
sudo pip3 install git+git://github.com/Teriks/pake@develop
```

CHAPTER 2

Module Doc

2.1 pake package

2.1.1 Module Contents

pake.**EXPORTS**

A dictionary object containing all current exports by name, you are free to modify this dictionary directly.

See: `pake.export()`, `pake.subpake()` and `pake.TaskContext.subpake()`.

Be careful and make sure it remains a dictionary object.

Export values must be able to **repr()** into parsable python literals.

pake.**init**(*args=None*, ***kwargs*)

Read command line arguments relevant to initialization, and return a `pake.Pake` object.

This function will print information to `pake.conf.stderr` and call `exit(pake.returncodes.BAD_ARGUMENTS)` immediately if arguments parsed from the command line or passed to the **args** parameter do not pass validation.

Parameters

- **args** – Optional command line arguments as an iterable, if not provided they will be parsed from the command line. This parameter is passed through `pake.util.handle_shell_args()`, so you may pass an arguments iterable containing nested non-string iterables, as well as plain values like Python integers if your specifying the **–jobs** argument for example.
- ****kwargs** – See below

Keyword Arguments

- **stdout** – Sets the value of `pake.Pake.stdout`
- **show_task_headers** (`bool`) – Sets the value of `pake.Pake.show_task_headers`

- **sync_output** (`bool`) – Sets the value of `pake.Pake.sync_output`, overriding the **–sync-output** option and the **PAKE_SYNC_OUTPUT** environmental variable. The default behavior of pake is to synchronize task output when tasks are running in parallel, unless it is overridden from the environment, command line, or here (in order of increasing override priority). Setting this value to **None** is the same as leaving it unspecified, no override will take place, and the default value of True, the environment, or the **–sync-output** specified value will be used in that order.

Raises `SystemExit` if bad command line arguments are parsed, or the **args** parameter contains bad arguments.

Returns `pake.Pake`

`pake.`**`de_init`**(*clear_conf=True*, *clear_exports=True*, *clear_env=True*)

Return the pake module to a pre-initialized state.

Used primarily for unit tests.

Defines read from STDIN via the **–stdin-defines** option are cached in memory after being read and not affected by this function, secondary calls to `pake.init()` will cause them to be read back from cache.

Parameters

- **`clear_conf`** – If **True**, call `pake.conf.reset()`.
- **`clear_exports`** – If **True**, call **clear** on `pake.EXPORTS`.
- **`clear_env`** – If **True**, clear any environmental variables pake.init has set.

`pake.`**`is_init`**()

Check if `pake.init()` has been called.

Returns True if `pake.init()` has been called.

`pake.`**`run`**(*pake_obj*, *tasks=None*, *jobs=None*, *call_exit=True*)

Run pake *(the program)* given a `pake.Pake` instance and default tasks.

This function should be used to invoke pake at the end of your pakefile.

This function will call **exit(return_code)** upon handling any exceptions from `pake.Pake.run()` or `pake.Pake.dry_run()` if **call_exit=True**, and will print information to `pake.Pake.stderr` if necessary.

This function will not call **exit** if pake executes successfully with a return code of zero.

This function will return pake's exit code when **call_exit=False**.

For all return codes see: `pake.returncodes`.

This function will never return `pake.returncodes.BAD_ARGUMENTS`, because `pake.init()` will have already called **exit**.

Raises `pake.PakeUninitializedException` if `pake.init` has not been called.

Raises `ValueError` if the **jobs** parameter is used, and is set less than 1.

Parameters

- **`pake_obj`** (`pake.Pake`) – A `pake.Pake` instance, created by `pake.init()`.
- **`tasks`** – A list of, or a single default task to run if no tasks are specified on the command line. Tasks specified on the command line completely override this argument.
- **`jobs`** – Call with an arbitrary number of max jobs, overriding the command line value of **–jobs**. The default value of this parameter is **None**, which means the command line value or default of 1 is not overridden.

- **call_exit** – Whether or not **exit(return_code)** should be called by this function on error. This defaults to **True** and when set to **False** the return code is instead returned to the caller.

Returns A return code from *pake.returncodes*.

pake.**terminate**(*pake_obj*, *return_code=0*)

Preform a graceful exit from a pakefile, printing the leaving directory or exit subpake message if needed, then exiting with a given return code. The default return code is *pake.returncodes.SUCCESS*.

This should be used as opposed to a raw **exit** call to ensure the output of pake remains consistent.

Use Case:

```
import os
import pake
from pake import returncodes

pk = pake.init()

# Say you need to wimp out of a build for some reason
# But not inside of a task.  pake.terminate will make sure the
# 'leaving directory/exiting subpake' message is printed
# if it needs to be.

if os.name == 'nt':
    pk.print('You really thought you could '
             'build my software on windows? nope!')

    pake.terminate(pk, returncodes.ERROR)

    # or

    # pk.terminate(returncodes.ERROR)


# Define some tasks...

@pk.task
def build(ctx):
     # You can use pake.terminate() inside of a task as well as exit()
     # pake.terminate() may offer more functionality than a raw exit()
     # in the future, however exit() will always work as well.

     something_bad_happened = True

     if something_bad_happened:
         pake.terminate(pk, returncodes.ERROR)

         # Or:

         # pk.terminate(returncodes.ERROR)

pake.run(pk, tasks=build)
```

pake.Pake.terminate() is a shortcut method which passes the **pake_obj** instance to this function for you.

Parameters

- **pake_obj** (pake.Pake) – A *pake.Pake* instance, created by *pake.init()*.

- **`return_code`** – Return code to exit the pakefile with, see *`pake.returncodes`* for standard return codes. The default return code for this function is *`pake.returncodes.SUCCESS`*. *`pake.returncodes.ERROR`* is intended to be used with **terminate** to indicate a generic error, but other return codes may be used.

Raises *`pake.PakeUninitializedException`* if *`pake.init`* has not been called.

`pake.`**`get_subpake_depth`**`()`

Get the depth of execution, which increases for nested calls to *`pake.subpake()`*

The depth of execution starts at 0.

Raises *`pake.PakeUninitializedException`* if *`pake.init`* has not been called.

Returns The current depth of execution (an integer >= 0)

`pake.`**`get_max_jobs`**`()`

Get the max number of jobs passed from the **–jobs** command line argument.

The minimum number of jobs allowed is 1.

Be aware, the value this function returns will not be affected by the optional **jobs** argument of *`pake.run()`* and *`pake.Pake.run()`*. It is purely for retrieving the value passed on the command line.

If you have overridden the command line or default job count using the **jobs** argument of the methods mentioned above, you can use the *`pake.Pake.max_jobs`* property on the *`pake.Pake`* instance returned by *`pake.init()`* to get the correct value inside of a task while pake is running.

Raises *`pake.PakeUninitializedException`* if *`pake.init`* has not been called.

Returns The max number of jobs from the **–jobs** command line argument. (an integer >= 1)

`pake.`**`get_init_file`**`()`

Gets the full path to the file *`pake.init()`* was called in.

Raises *`pake.PakeUninitializedException`* if *`pake.init`* has not been called.

Returns Full path to pakes entrypoint file, or **None**

`pake.`**`get_init_dir`**`()`

Gets the full path to the directory pake started running in.

If pake preformed any directory changes, this returns the working path before that happened.

Raises *`pake.PakeUninitializedException`* if *`pake.init`* has not been called.

Returns Full path to init dir, or **None**

`pake.`**`export`**`(`*name*, *value*`)`

Exports a define that can be retrieved in subpake scripts via *`pake.Pake.get_define()`*.

This function can redefine the value of an existing export as well.

The *`pake.EXPORTS`* dictionary can also be manipulated directly.

Export values must be able to **repr()** into parsable python literals.

Parameters

- **`name`** – The name of the define.
- **`value`** – The value of the define.

`pake.`**`subpake`**`(`**args*, *stdout=None*, *silent=False*, *ignore_errors=False*, *call_exit=True*, *readline=True*, *collect_output=False*, *collect_output_lock=None*`)`

Execute a `pakefile.py` script, changing directories if necessary.

This function should not be used inside tasks, use: *pake.TaskContext.subpake()* instead. A *pake.TaskContext()* instance is passed into the single parameter of each task, usually named **ctx**.

pake.subpake() allows similar syntax to *pake.TaskContext.call()* for its ***args** parameter.

Subpake scripts do not inherit the **–jobs** argument from the parent script, if you want to run them with multithreading enabled you need to pass your own **–jobs** argument manually.

Example:

```
# These are all equivalent

pake.subpake('dir/pakefile.py', 'task_a', '-C', 'some_dir')

pake.subpake(['dir/pakefile.py', 'task_a', '-C', 'some_dir'])

# note the nested iterable containing string arguments

pake.subpake(['dir/pakefile.py', 'task_a', ['-C', 'some_dir']])

pake.subpake('dir/pakefile.py task_a -C some_dir')
```

Parameters

- **args** – The script, and additional arguments to pass to the script. You may pass the command words as a single iterable, a string, or as variadic arguments.
- **stdout** – The file output to write all of the pakefile's output to. (defaults to `pake.conf.stdout`) The pakefile's **stderr** will be redirected to its **stdout**, so the passed file object will receive all output from the pakefile including error messages.
- **silent** – Whether or not to silence all output from the subpake script.
- **ignore_errors** – If this is **True**, this function will never call **exit** or throw *pake.SubpakeException* if the executed pakefile returns with a non-zero exit code. It will instead return the exit code from the subprocess to the caller.
- **call_exit** – Whether or not to print to `pake.conf.stderr` and immediately call **exit** if the pakefile script encounters an error. The value of this parameter will be disregarded when **ignore_errors=True**.
- **readline** – Whether or not to use **readline** for reading process output when **ignore_errors** and **silent** are **False**, this is necessary for live output in that case. When live output to a terminal is not required, such as when writing to a file on disk, setting this parameter to **False** results in more efficient writes. This parameter defaults to **True**
- **collect_output** – Whether or not to collect all subpake output to a temporary file and then write it incrementally to the **stdout** parameter when the process finishes. This can help prevent crashes when dealing with lots of output. When you pass **True** to this parameter, the **readline** parameter is ignored. See: *Output synchronization with ctx.call & ctx.subpake*
- **collect_output_lock** – If you provide a lockable object such as `threading.Lock` or `threading.RLock`, The subpake function will try to acquire the lock before incrementally writing to the **stdout** parameter when **collect_output=True**. The lock you pass is only required to implement a context manager and be usable in a **with** statement, no methods are called on the lock. *pake.TaskContext.subpake()* will pass *pake.TaskContext.io_lock* for you if **collect_output=True**.

Raises `ValueError` if no command + optional command arguments are provided.

Raises `FileNotFoundError` if the first argument *(the pakefile)* is not found.

Raises *pake.SubpakeException* if the called pakefile script encounters an error and the following is true: **exit_on_error=False** and **ignore_errors=False**.

class pake.**Pake**(*stdout=None*, *sync_output=True*, *show_task_headers=True*)

Bases: `object`

Pake's main instance, which should not be initialized directly.

Use: *pake.init()* to create a *pake.Pake* object and initialize the pake module.

stdout

The file object that task output gets written to, as well as 'changing directory/entering & leaving subpake' messages. If you set this, make sure that you set it to an actual file object that implements **fileno()**. `io.StringIO` and pseudo file objects with no **fileno()** will not work with all of pake's subprocess spawning functions.

This attribute can be modified directly.

sync_output

Whether or not the pake instance should queue task output and write it in a synchronized fashion when running with more than one job. This defaults to **True** unless the environmental variable `PAKE_SYNC_OUTPUT` is set to **0**, or the command line option **–output-sync False** is specified.

If this is disabled (Set to **False**), task output may become interleaved and scrambled when running pake with more than one job. Pake will run somewhat faster however.

This attribute can be modified directly.

show_task_headers

Whether or not pake should print **Executing Task:** headers for tasks that are about to execute, the default value is **True**. If you set this to **False** task headers will be disabled for all tasks except ones that explicitly specify **show_header=True**. See the **show_header** parameter of *pake.Pake.task()* and *pake.Pake.add_task()*, which allows you to disable or force enable the task header for a specific task.

This attribute can be modified directly.

add_task(*name*, *func*, *dependencies=None*, *inputs=None*, *outputs=None*, *show_header=None*)

Method for programmatically registering pake tasks.

This method expects for the most part the same argument types as the *pake.Pake.task()* decorator.

Example:

```
# A contrived example using a callable class

class FileToucher:
    """Task Documentation Here"""

    def __init__(self, tag):
        self._tag = tag

    def __call__(self, ctx):
        ctx.print('Toucher {}'.format(self._tag))

        fp = pake.FileHelper(ctx)

        for i in ctx.outputs:
            fp.touch(i)


task_instance_a = FileToucher('A')
```

(continues on next page)

(continued from previous page)

```
task_instance_b = FileToucher('B')

pk.add_task('task_a', task_instance_a, outputs=['file_1', 'file_2'])

pk.add_task('task_b', task_instance_b, dependencies=task_instance_a, outputs=
↪'file_3')

# Note: you can refer to dependencies by name (by string) as well as␣
↪reference.

# Equivalent calls:

# pk.add_task('task_b', task_instance_b, dependencies='task_a', outputs='file_
↪3')
# pk.add_task('task_b', task_instance_b, dependencies=['task_a'], outputs=
↪'file_3')
# pk.add_task('task_b', task_instance_b, dependencies=[task_instance_a],␣
↪outputs='file_3')


# Example using a function

def my_task_func_c(ctx):
    ctx.print('my_task_func_c')

pk.add_task('task_c', my_task_func_c, dependencies='task_b')

pake.run(pk, tasks=my_task_func_c)

# Or equivalently:

# pake.run(pk, tasks='task_c')
```

Parameters

- **`name`** – The name of the task
- **`func`** – The task function (or callable class)
- **`dependencies`** – List of task dependencies or single task, by name or by reference
- **`inputs`** – List of input files/directories, or a single input (accepts input file generators like *`pake.glob()`*)
- **`outputs`** – List of output files/directories, or a single output (accepts output file generators like *`pake.pattern()`*)
- **`show_header`** – Whether or not to print an **Executing Task:** header when the task begins executing. This defaults to **None**, which means the header is printed unless `pake.Pake.show_task_header` is set to **False**. If you specify **True** and `pake.Pake.show_task_header` is set to **False**, it will force the task header to print anyway. By explicitly specifying **True** you override `pake.Pake.show_task_header`.

Returns The *`pake.TaskContext`* for the new task.

`dry_run`(*tasks*)

Dry run over task, print a 'visited' message for each visited task.

When using change detection, only out of date tasks will be visited.

Raises `ValueError` If **tasks** is **None** or an empty list.

Raises *`pake.MissingOutputsException`* if a task defines input files/directories without specifying any output files/directories.

Raises *`pake.InputNotFoundException`* if a task defines input files/directories but one of them was not found on disk.

Raises *`pake.UndefinedTaskException`* if one of the default tasks given in the **tasks** parameter is unregistered.

Parameters **`tasks`** – Single task, or Iterable of task functions to run (by ref or name).

`get_define`(*name*, *default=None*)

Get a defined value.

This is used to get defines off the command line, as well as retrieve values exported from top level pake scripts.

If the define is not found, then **None** is returned by default.

The indexer operator can also be used on the pake instance to fetch defines, IE:

```
import pake

pk = pake.init()

value = pk['YOURDEFINE']
```

Which also produces **None** if the define does not exist.

See: *Specifying define values* for documentation covering how to specify defines on the command line, as well as what types of values you can use for your defines.

Parameters

- **`name`** – Name of the define
- **`default`** – The default value to return if the define does not exist

Returns The defines value as a python literal corresponding to the defines type.

`get_task_context`(*task*)

Get the *`pake.TaskContext`* object for a specific task.

Raises `ValueError` if the **task** parameter is not a string or a callable function/object.

Raises *`pake.UndefinedTaskException`* if the task in not registered.

Parameters **`task`** – Task function or function name as a string

Returns *`pake.TaskContext`*

`get_task_name`(*task*)

Returns the name of a task by the task function or callable reference used to define it.

The name of the task may be different than the name of the task function/callable when *`pake.Pake.add_task()`* is used to register the task.

If a string is passed it is returned unmodified as long as the task exists, otherwise a *`pake.UndefinedTaskException`* is raised.

Example:

```
@pk.task
def my_task(ctx):
    pass

def different_name(ctx):
    pass

pk.add_task("my_task2", different_name)

pk.get_task_name(my_task) # -> "my_task"

pk.get_task_name(different_name) # -> "my_task2"
```

Parameters `task` – Task name string, or registered task callable.

Raises `ValueError` if the **task** parameter is not a string or a callable function/object.

Raises *pake.UndefinedTaskException* if the task function/callable is not registered to the pake context.

Returns Task name string.

has_define(*name*)

Test if a define with a given name was provided to pake.

This is useful if **None** might be a valid value for your define, and you just want to know if it was actually specified on the command line or with **–stdin-defines**.

Parameters `name` – The name of the define.

Returns **True** if a define with the given name exists.

is_running

Check if pake is currently running tasks.

This can be used to determine if code is executing inside of a task.

Example:

```
import pake

pk = pake.init()

pk.print(pk.is_running) # -> False

@pk.task
def my_task(ctx):
    ctx.print(pk.is_running) # -> True


pake.run(pk, tasks=my_task, call_exit=False)

pk.print(pk.is_running) # -> False
```

Returns **True** if pake is currently running tasks, **False** otherwise.

max_jobs

Returns the value of the **jobs** parameter used in the last invocation of *pake.Pake.run()*.

This can be used inside of a task to determine if pake is running in multithreaded mode, and the maximum amount of threads it has been allowed to use for the current invocation.

A **max_jobs** value of **1** indicates that pake is running all tasks in the current thread, anything greater than **1** means pake is sending tasks to a threadpool.

See Also: *`pake.Pake.threadpool`*

`merge_defines_dict`(*dictionary*)

Merge the current defines with another dictionary, overwriting anything that is already defined with the value from the new dictionary.

Parameters **`dictionary`** – The dictionary to merge into the current defines.

`print`(**args*, ***kwargs*)

Print to the file object assigned to *`pake.Pake.stdout`*

Shorthand for: `print(..., file=pk.stdout)`

`run`(*tasks*, *jobs=1*)

Run all given tasks, with an optional level of concurrency.

Raises `ValueError` if **jobs** is less than 1, or if **tasks** is **None** or an empty list.

Raises *`pake.TaskException`* if an exception occurred while running a task.

Raises *`pake.TaskExitException`* if `SystemExit` or an exception derived from it such as *`pake.TerminateException`* is raised inside of a task.

Raises *`pake.MissingOutputsException`* if a task defines input files/directories without specifying any output files/directories.

Raises *`pake.InputNotFoundException`* if a task defines input files/directories but one of them was not found on disk.

Raises *`pake.UndefinedTaskException`* if one of the default tasks given in the *tasks* parameter is unregistered.

Parameters

- **`tasks`** – Single task, or Iterable of task functions to run (by ref or name).
- **`jobs`** – Maximum number of threads, defaults to 1. (must be >= 1)

`run_count`

Contains the number of tasks ran/visited by the last invocation of *`pake.Pake.run()`* or *`pake.Pake.dry_run()`*

If a task did not run because change detection decided it did not need to, then it does not count towards this total. This also applies when doing a dry run with *`pake.Pake.dry_run()`*

Returns Number of tasks last run.

`set_define`(*name*, *value*)

Set a defined value.

Parameters

- **`name`** – The name of the define.
- **`value`** – The value of the define.

`set_defines_dict`(*dictionary*)

Set and overwrite all defines with a dictionary object.

Parameters **`dictionary`** – The dictionary object

task(**args*, *i=None*, *o=None*, *show_header=None*)

Decorator for registering pake tasks.

Any input files specified must be accompanied by at least one output file.

A callable object, or list of callable objects may be passed to **i** or **o** in addition to a raw file/directory name or names. This is how **pake.glob** and **pake.pattern** work.

Input/Output Generation Example:

```python
def gen_inputs(pattern):
    def input_generator():
        return glob.glob(pattern)
    return input_generator

def gen_output(pattern):
    def output_generator(inputs):
        # inputs is always a flat list, and a copy
        # inputs is safe to mutate if you want

        for inp in inputs:
            dir = os.path.dirname(inp)
            name, ext = os.path.splitext(os.path.basename(inp))
            yield pattern.replace('{dir}', dir).replace('%', name).replace('
↪{ext}', ext)

    return output_generator

@pk.task(i=gen_inputs('*.c'), o=gen_outputs('%.o'))
def my_task(ctx):
    # Do your build task here
    pass


@pk.task(i=[gen_inputs('src_a/*.c'), gen_inputs('src_b/*.c')], o=gen_outputs('
↪{dir}/%.o'))
def my_task(ctx):
    # Do your build task here
    pass
```

Dependencies Only Example:

```python
@pk.task(dependency_task_a, dependency_task_b)
def my_task(ctx):
    # Do your build task here
    pass
```

Change Detection Examples:

```python
# Dependencies come before input and output files.

@pk.task(dependency_task_a, dependency_task_b, i='main.c', o='main')
def my_task(ctx):
    # Do your build task here
    pass


# Tasks without input or output files will always run when specified.

```

(continues on next page)

(continued from previous page)

```
@pk.task
def my_task(ctx):
    # I will always run when specified!
    pass


# Tasks with dependencies but no input or output files will also
# always run when specified.

@pk.task(dependency_task_a, dependency_task_b)
def my_task(ctx):
    # I will always run when specified!
    pass


# Having an output with no input is allowed, this task
# will always run.  The opposite (having an input file with no output file)
# will cause an error.  ctx.outdated_outputs is populated with 'main' in this␣
↪case.

@pk.task(o='main')
def my_task(ctx):
    # Do your build task here
    pass


# Single input and single output, 'main.c' has its creation time checked
# against 'main'

@pk.task(i='main.c', o='main')
def my_task(ctx):
    # Do your build task here
    pass


# When multiple input files exist and there is only one output file, each␣
↪input file
# has its creation time checked against the output files creation time.

@pk.task(i=['otherstuff.c', 'main.c'], o='main')
def my_task(ctx):
    # Do your build task here
    pass


# all files in 'src/*.c' have their creation date checked against 'main'

@pk.task(i=pake.glob('src/*.c'), o='main')
def my_task(ctx):
    # Do your build task here
    pass


# each input file has its creation date checked against its corresponding
# output file in this case.  Out of date file names can be found in
# ctx.outdated_inputs and ctx.outdated_outputs.  ctx.outdated_pairs is a
# convenience property which returns: zip(ctx.outdated_inputs, ctx.outdated_
↪outputs)
```

(continues on next page)

(continued from previous page)

```
@pk.task(i=['file_b.c', 'file_b.c'], o=['file_b.o', 'file_b.o'])
def my_task(ctx):
    # Do your build task here
    pass


# Similar to the above, inputs and outputs end up being of the same
# length when using pake.glob with pake.pattern

@pk.task(i=pake.glob('src/*.c'), o=pake.pattern('obj/%.o'))
def my_task(ctx):
    # Do your build task here
    pass


# All input files have their creation date checked against all output files
# if there are more inputs than outputs, in general.

@pk.task(i=['a.c', 'b.c', 'c.c'], o=['main', 'what_are_you_planning'])
def my_task(ctx):
    # Do your build task here
    pass


# Leaving the inputs and outputs as empty list will cause the task
# to never run.

@pk.task(i=[], o=[])
def my_task(ctx):
    # I will never run!
    pass


# If an input generator produces no results
# (IE, something like pake.glob returns no files) and the tasks
# outputs also end up being empty such as in this case,
# then the task will never run.

@pk.task(i=pake.glob('*.some_extension_that_no_file_has'), o=pake.pattern('%.o
↪'))
def my_task(ctx):
    # I will never run!
    pass
```

Raises *pake.UndefinedTaskException* if a given dependency is not a registered task function.

Parameters

- **args** – Tasks which this task depends on, this may be passed as variadic arguments or a single iterable object.
- **i** – Optional input files/directories for change detection.
- **o** – Optional output files/directories for change detection.
- **show_header** – Whether or not to print an **Executing Task:** header when the task

begins executing. This defaults to **None**, which means the header is printed unless `pake.Pake.show_task_header` is set to **False**. If you specify **True** and `pake.Pake.show_task_header` is set to **False**, it will force the task header to print anyway. By explicitly specifying **True** you override the `pake.Pake.show_task_header` setting.

task_contexts
Retrieve the task context objects for all registered tasks.

Returns List of *pake.TaskContext*.

task_count
Returns the number of registered tasks.

Returns Number of tasks registered to the *pake.Pake* instance.

terminate(*return_code=0*)
Shorthand for `pake.terminate(this, return_code=return_code)`.

See for more details: *pake.terminate()*

Parameters **return_code** – Return code to exit the pakefile with. The default return code is *pake.returncodes.SUCCESS*.

threadpool
Current execution thread pool.

This will never be anything other than **None** unless pake is running, and its max job count is greater than 1.

Pake is considered to be running when *pake.Pake.is_running* equals **True**.

If pake is running with a job count of 1, no threadpool is used so this property will be **None**.

class pake.**TaskContext**(*pake_obj*, *node*)
Bases: `object`

Contextual object passed to each task.

inputs

All file inputs, or an empty list.

Note:

Not available outside of a task, may only be used while a task is executing.

outputs

All file outputs, or an empty list.

Note:

Not available outside of a task, may only be used while a task is executing.

outdated_inputs

All changed file inputs (or inputs who's corresponding output is missing), or an empty list.

Note:

Not available outside of a task, may only be used while a task is executing.

oudated_outputs

All out of date file outputs, or an empty list

Note:

Not available outside of a task, may only be used while a task is executing.

call(**args*, *stdin=None*, *shell=False*, *ignore_errors=False*, *silent=False*, *print_cmd=True*, *collect_output=False*)

Calls a sub process and returns its return code.

all **stdout/stderr** is written to the task IO file stream. The full command line which was used to start the process is printed to the task IO queue before the output of the command, unless **print_cmd=False**.

You can prevent the process from sending its **stdout/stderr** to the task IO queue by specifying **silent=True**

If a process returns a non-zero return code, this method will raise `pake.TaskSubprocessException` by default.

If you want the value of non-zero return codes to be returned then you must pass **ignore_errors=True** to prevent `pake.TaskSubprocessException` from being thrown, or instead catch the exception and get the return code from it.

Note:

`pake.TaskSubprocessException.output_stream` will be available for retrieving the output of the process (**stdout** and **stderr** combined) if you handle the exception, the file stream will be a text mode file object at **seek(0)**.

Example:

```
# strings are parsed using shlex.parse

ctx.call('gcc -c test.c -o test.o')

ctx.call('gcc -c {} -o {}'.format('test.c', 'test.o'))

# pass the same command as a list

ctx.call(['gcc', '-c', 'test.c', '-o', 'test.o'])

# pass the same command using the variadic argument *args

ctx.call('gcc', '-c', 'test.c', '-o', 'test.o')

# non string iterables in command lists will be flattened,
# allowing for this syntax to work.  ctx.inputs and ctx.outputs
# are both list objects, but anything that is iterable will work

ctx.call(['gcc', '-c', ctx.inputs, '-o', ctx.outputs])


# Fetch a non-zero return code without a
# pake.TaskSubprocessException.  ctx.check_call
# is better used for this task.

code = ctx.call('which', 'am_i_here',
                ignore_errors=True,  # Ignore errors (non-zero return codes)
                print_cmd=False,    # Don't print the command line executed
                silent=True)  # Don't print stdout/stderr to task IO
```

Parameters

- **args** – The process command/executable, and additional arguments to pass to the process. You may pass the command words as a single iterable, a string, or as variadic arguments.
- **stdin** – Optional file object to pipe into the called process's **stdin**.
- **shell** – Whether or not to use the system shell for execution of the command.
- **ignore_errors** – Whether or not to raise a `pake.TaskSubprocessException` on non-zero exit codes.
- **silent** – Whether or not to silence **stdout/stderr** from the command. This does keep pake from printing what command line was run, see the **print_cmd** argument for that.
- **print_cmd** – Whether or not to print the executed command line to the tasks output. The **silent** argument will not keep pake from printing the executed command, only this argument can do that.
- **collect_output** – Whether or not to collect all process output to a temporary file and then incrementally write it back to `pake.TaskContext.io` in a synchronized fashion, so that all command output is guaranteed to come in order and not become interleaved with the output of other tasks when using `pake.TaskContext.multitask()`. See: *Output synchronization with ctx.call & ctx.subpake*

Returns The process return code.

Raises `pake.TaskSubprocessException` if *ignore_errors* is *False* and the process exits with a non-zero exit code.

Raises `OSError` (commonly) if a the executed command or file does not exist. This exception will still be raised even if **ignore_errors** is **True**.

Raises `ValueError` if no command + optional arguments are provided.

static check_call(**args*, *stdin=None*, *shell=False*, *ignore_errors=False*)

Get the return code of an executed system command, without printing any output to the tasks IO queue by default.

None of the process's **stdout/stderr** will go to the task IO queue, and the command that was run will not be printed either.

This function raises `pake.TaskSubprocessException` on non-zero return codes by default.

You should pass pass **ignore_errors=True** if you want this method to return the non-zero value, or instead catch the exception and get the return code from it.

Note:

`pake.TaskSubprocessException.output` and `pake.TaskSubprocessException.output_stream` will **not** be available in the exception if you handle it.

Raises `pake.TaskSubprocessException` if **ignore_errors** is **False** and the process exits with a non-zero return code.

Raises `OSError` (commonly) if a the executed command or file does not exist. This exception will still be raised even if **ignore_errors** is **True**.

Raises `ValueError` if no command + optional arguments are provided.

Parameters

- **args** – Command arguments, same syntax as `pake.TaskContext.call()`
- **stdin** – Optional file object to pipe into the called process's **stdin**.
- **shell** – Whether or not to use the system shell for execution of the command.

- **ignore_errors** – Whether to ignore non-zero return codes and return the code anyway.

Returns Integer return code.

static check_output (**args*, *stdin=None*, *shell=False*, *ignore_errors=False*)

Return the output of a system command as a bytes object, without printing its **stdout/stderr** to the task IO queue. The process command line that was run will not be printed either.

The returned bytes output will include **stdout** and **stderr** combined, and it can be decoded into a string by using the **decode**() method on pythons built in **bytes** object.

This function raises `pake.TaskSubprocessException` on non-zero return codes by default.

If you want to return possible error output from the called process's **stderr** you should pass **ignore_errors=True**, or instead catch the exception and get the process output from it.

Note:

`pake.TaskSubprocessException.output` will be available for retrieving the output of the process if you handle the exception, the value will be all of **stdout/stderr** as a **bytes** object that must be decoded into a string.

Raises `pake.TaskSubprocessException` if **ignore_errors** is False and the process exits with a non-zero return code.

Raises `OSError` (commonly) if a the executed command or file does not exist. This exception will still be raised even if **ignore_errors** is **True**.

Raises `ValueError` if no command + optional arguments are provided.

Parameters

- **args** – Command arguments, same syntax as `pake.TaskContext.call()`
- **stdin** – Optional file object to pipe into the called process's **stdin**.
- **shell** – Whether or not to use the system shell for execution of the command.
- **ignore_errors** – Whether to ignore non-zero return codes and return the output anyway.

Returns Bytes object (program output data)

dependencies

Immediate dependencies of this task.

returns a list of `pake.TaskContext` representing each immediate dependency of this task.

Note:

This property **will** return a meaningful value outside of a task.

dependency_outputs

Returns a list of output files/directories which represent the outputs of the tasks immediate dependencies.

Note:

Not available outside of a task, may only be used while a task is executing.

func

Task function reference.

This function will be an internal wrapper around the one you specified and you should not call it.

There is not currently a way to get a reference to your actual unwrapped task function from the `pake.Pake` object or elsewhere.

However since the `functools.wraps()` decorator is used when wrapping your task function, metadata such as **func.__doc__** will be maintained on this function reference.

`io`

The task IO file stream, a file like object that is only open for writing during a tasks execution.

Any output to be displayed for the task should be written to this file object.

This file object is a text mode stream, it can be used with the built in **print** function and other methods that can write text data to a file like object.

When you run pake with more than one job, this will be a reference to a temporary file unless *`pake.Pake.sync_output`* is **False** (It is **False** when **–no-sync-output** is used on the command line).

The temporary file queues up task output when in use, and the task context acquires a lock and writes it incrementally to *`pake.Pake.stdout`* when the task finishes. This is done to avoid having concurrent task's writing interleaved output to *`pake.Pake.stdout`*.

If you run pake with only 1 job or *`pake.Pake.sync_output`* is **False**, this property will return a direct reference to *`pake.Pake.stdout`*.

`io_lock`

A contextual lock for acquiring exclusive access to *`pake.TaskContext.io`*.

This context manager acquires an internal lock for *`pake.Pake.stdout`* that exists on the *`pake.Pake`* object when *`pake.Pake.max_jobs`* is **1**.

Otherwise, it will acquire a lock for *`pake.TaskContext.io`* that exists inside of the task context, since the task will be buffering output to an individual temporary file when *`pake.Pake.max_jobs`* is greater than **1**.

If *`pake.Pake.sync_output`* is **False**, the context manager returned by this property will not attempt to acquire any lock.

Returns A context manager object that can be used in a **with** statement.

`multitask`(*aggregate_exceptions=False*)

Returns a contextual object for submitting work to pake's current thread pool.

```
@pk.task(i=pake.glob('src/*.c'), o=pake.pattern('obj/%.o'))
    def build_c(ctx):
        with ctx.multitask() as mt:
            for i, o in ctx.outdated_pairs:
                mt.submit(ctx.call, ['gcc', '-c', i, '-o', o])
```

At the end of the **with** statement, all submitted tasks are simultaneously waited on.

The tasks will be checked in order of submission for exceptions, if an exception is found then the default behavior is to re-raise it on the foreground thread.

You can specify **aggregate_exceptions=True** if you want all of the exceptions to be collected into a *`pake.AggregateException`*, which will then be raised when *`pake.MultitaskContext.shutdown()`* is called with **wait=True**.

shutdown is called at the end of your **with** statement with the **wait** parameter set to **True**.

Parameters **`aggregate_exceptions`** – Whether or not the returned executor should collect exceptions from all tasks that ended due to an exception, and then raise a *`pake.AggregateException`* containing them upon shutdown.

Returns *`pake.MultitaskContext`*

`name`

The task name.

Returns The task name, as a string.

node
The *pake.TaskGraph* node for the task.

outdated_pairs
Short hand for: `zip(ctx.outdated_inputs, ctx.outdated_outputs)`

Returns a generator object over outdated (input, output) pairs.

This is only useful when the task has the same number of inputs as it does outputs.

Example:

```
@pk.task(i=pake.glob('src/*.c'), o=pake.pattern('obj/%.o'))
def build_c(ctx):
    for i, o in ctx.outdated_pairs:
        ctx.call(['gcc', '-c', i, '-o', o])
```

Note:

Not available outside of a task, may only be used while a task is executing.

pake
The *pake.Pake* instance the task is registered to.

print(**args*, ***kwargs*)
Prints to the task IO file stream using the builtin print function.

Shorthand for: `print(..., file=ctx.io)`

subpake(**args*, *silent=False*, *ignore_errors=False*, *collect_output=False*)
Run *pake.subpake()* and direct all output to the task IO file stream.

Parameters

- **args** – The script, and additional arguments to pass to the script. You may pass a list, or use variadic arguments.
- **silent** – If **True**, avoid printing output from the sub-pakefile to the tasks IO queue.
- **ignore_errors** – If this is **True**, this function will not throw *pake.SubpakeException* if the executed pakefile returns with a non-zero exit code. It will instead return the exit code from the subprocess to the caller.
- **collect_output** – Whether or not to collect all subpake output to a temporary file and then incrementally write it back to *pake.TaskContext.io* in a synchronized fashion, so that all command output is guaranteed to come in order and not become interleaved with the output of other tasks when using *pake.TaskContext.multitask()*. See: *Output synchronization with ctx.call & ctx.subpake*

Raises `ValueError` if no command + optional command arguments are provided.

Raises `FileNotFoundError` if the first argument (the pakefile) is not found.

Raises *pake.SubpakeException* if the called pakefile script encounters an error and **ignore_errors=False** .

class pake.**MultitaskContext**(*ctx*, *aggregate_exceptions=False*)
Bases: `concurrent.futures._base.Executor`

Returned by *pake.TaskContext.multitask()* (see for more details).

This object has (for the most part) has the exact same behavior and interface as `concurrent.futures.ThreadPoolExecutor` from the built in Python module `concurrent.futures`.

If you need further reference on how its member functions behave, you can also consult the official Python doc for that class.

This object is meant to be used in a **with** statement. At the end of the **with** statement all of your submitted work will be waited on, so you do not have to do it manually with `pake.MultitaskContext.shutdown()`.

Using a **with** statement is also exception safe.

`aggregate_exceptions`

Whether or not the multitasking context should collect all exceptions that occurred inside of submitted tasks upon shutdown, and then raise a `pake.AggregateException` containing them.

This is **False** by default, the normal behaviour is to search the tasks in the order of submission for exceptions upon shutdown, and then re-raise the first exception that was encountered on the foreground thread.

`map`(*fn*, **iterables*, *timeout=None*, *chunksize=1*)

Returns an iterator equivalent to `map(fn, iter)`.

Parameters

- **`fn`** – A callable that will take as many arguments as there are passed iterables.
- **`timeout`** – The maximum number of seconds to wait. If **None**, then there is no limit on the wait time.
- **`chunksize`** – The size of the chunks the iterable will be broken into.

Returns An iterator equivalent to: `map(func, *iterables)` but the calls may be evaluated out-of-order.

Raises `TimeoutError` If the entire result iterator could not be generated before the given timeout.

Raises `Exception` If `fn(*args)` raises for any values.

`shutdown`(*wait=True*)

Shutdown multitasking and free resources, optionally wait on all submitted tasks.

It is not necessary to call this function if you are using the context in a **with** statement.

If you specify **wait=False**, this method will not propagate any exceptions out of your submitted tasks.

Parameters **`wait`** – Whether or not to wait on all submitted tasks, default is true.

`submit`(*fn*, **args*, ***kwargs*)

Submit a task to pakes current threadpool.

If no thread pool exists, such as in the case of **–jobs 1**, then the submitted function is immediately executed in the current thread.

This function has an identical call syntax to **concurrent.futures.Executor.submit**.

Example:

```
mt.submit(work_function, arg1, arg2, keyword_arg='arg')
```

Returns `concurrent.futures.Future`

class pake.**TaskGraph**(*name*, *func*)

Bases: `pake.graph.Graph`

Task graph node.

func
Task function reference.

This function will be an internal wrapper around the one you specified and you should not call it.

There is not currently a way to get a reference to your actual unwrapped task function from the `pake.Pake` object or elsewhere.

However since the `functools.wraps()` decorator is used when wrapping your task function, metadata such as **func.__doc__** will be maintained on this function reference.

add_edge(*edge*)
Add an edge to the graph.

Parameters **edge** – The edge to add (another `pake.graph.Graph` object)

edges
Retrieve a set of edges from this graph node.

Returns A set() of adjacent nodes.

name
The task name.

Returns The task name, as a string.

remove_edge(*edge*)
Remove an edge from the graph by reference.

Parameters **edge** – Reference to a `pake.graph.Graph` object.

topological_sort()
Return a generator object that runs topological sort as it is iterated over.

Nodes that have been visited will not be revisited, making infinite recursion impossible.

Returns A generator that produces `pake.graph.Graph` nodes.

pake.**pattern**(*file_pattern*)
Produce a substitution pattern that can be used in place of an output file.

The **%** character represents the file name, while **{dir}** and **{ext}** represent the directory of the input file, and the input file extension.

Example:

```
@pk.task(i=pake.glob('src/*.c'), o=pake.pattern('obj/%.o'))
def build_c(ctx):
    for i, o in ctx.outdated_pairs:
        ctx.call('gcc', '-c', i, '-o', o)

@pk.task(i=[pake.glob('src_a/*.c'), pake.glob('src_b/*.c')], o=pake.pattern('{dir}
↪/%.o'))
def build_c(ctx):
    for i, o in ctx.outdated_pairs:
        ctx.call('gcc', '-c', i, '-o', o)
```

`pake.pattern()` returns function similar to this:

```
def output_generator(inputs):
    # inputs is always a flat list, and a copy
    # inputs is safe to mutate

```

(continues on next page)

(continued from previous page)

```
    for inp in inputs:
        dir = os.path.dirname(inp)
        name, ext = os.path.splitext(os.path.basename(inp))
        yield file_pattern.replace('{dir}', dir).replace('%', name).replace('{ext}
↪', ext)
```

pake.**glob**(*expression*)

Deferred file input glob, the glob is not executed until the task executes.

This input generator handles recursive directory globs by default, denoted by a double asterisk.

It will return directory names as well if your glob expression matches them.

The syntax used is the same as the built in `glob.glob()` from pythons `glob` module.

Example:

```
@pk.task(build_c, i=pake.glob('obj/*.o'), o='main')
def build_exe(ctx):
   ctx.call('gcc', ctx.inputs, '-o', ctx.outputs)

@pk.task(build_c, i=[pake.glob('obj_a/*.o'), pake.glob('obj_b/*.o')], o='main')
def build_exe(ctx):
   ctx.call('gcc', ctx.inputs, '-o', ctx.outputs)
```

Recursive Directory Search Example:

```
# Find everything under 'src' that is a .c file, including
# in sub directories of 'src' and all the way to the bottom of
# the directory tree.

# pake.pattern is used to put the object file for each .c file
# next to it in the same directory.

@pk.task(i=pake.glob('src/**/*.c'), o=pake.pattern('{dir}/%.o'))
def build_c(ctx):
    for i, o in ctx.outdated_pairs:
        ctx.call('gcc', '-c', i, '-o', o)
```

pake.glob() returns a function similar to this:

```
def input_generator():
    return glob.iglob(expression, recursive=True)
```

Returns A callable function object, which returns a generator over the file glob results as strings.

class pake.**FileHelper**(*printer=None*)

Bases: `object`

A helper class for dealing with common file operations inside and outside of pake tasks. Instantiating this class with the **printer** parameter set to a *pake.TaskContext* instance will cause it to print information about file system operations it performs to the tasks output queue. Each function can be silenced by setting the **silent** parameter of the function to **True**.

copy(*src*, *dest*, *copy_metadata=False*, *follow_symlinks=True*, *silent=False*)

Copy a file to a destination.

See `shutil.copy()` and `shutil.copy2()` (when **copy_metadata** is True)

Parameters

- **`src`** – The file.
- **`dest`** – The destination path.
- **`copy_metadata`** – If True, file metadata like creation time will be copied to the new file.
- **`follow_symlinks`** – Whether or not to follow symlinks while copying.
- **`silent`** – If True, Don't print information to the tasks output.

`copytree`(*self*, *src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=shutil.copy2*, *ignore_dangling_symlinks=False*, *silent=False*)

Recursively copy a directory tree, See `shutil.copytree()`.

The destination directory must not already exist. If exception(s) occur, an Error is raised with a list of reasons.

If the optional symlinks flag is true, symbolic links in the source tree result in symbolic links in the destination tree; if it is false, the contents of the files pointed to by symbolic links are copied. If the file pointed by the symlink doesn't exist, an exception will be added in the list of errors raised in an Error exception at the end of the copy process.

You can set the optional ignore_dangling_symlinks flag to true if you want to silence this exception. Notice that this has no effect on platforms that don't support `os.symlink()`.

The optional ignore argument is a callable. If given, it is called with the *src* parameter, which is the directory being visited by `shutil.copytree()`, and *names* which is the list of *src* contents, as returned by `os.listdir()`:

callable(src, names) -> ignored_names

Since `shutil.copytree()` is called recursively, the callable will be called once for each directory that is copied. It returns a list of names relative to the *src* directory that should not be copied.

The optional copy_function argument is a callable that will be used to copy each file. It will be called with the source path and the destination path as arguments. By default, `shutil.copy2()` is used, but any function that supports the same signature (like `shutil.copy()`) can be used.

Raises **`shutil.Error`** – If exception(s) occur, an Error is raised with a list of reasons.

Parameters

- **`src`** – The source directory tree.
- **`dst`** – The destination path.
- **`symlinks`** – If True, try to copy symlinks.
- **`ignore`** – Callable, used specifying files to ignore in a specific directory as copytree walks the source directory tree. `callable(src, names) -> ignored_names`
- **`copy_function`** – The copy function, if not specified `shutil.copy2()` is used.
- **`ignore_dangling_symlinks`** – If True, don't throw an exception when the file pointed to by a symlink does not exist.
- **`silent`** – If True, Don't print info the the tasks output.

`glob_remove`(*glob_pattern*, *silent=False*)

Remove files using a glob pattern, this makes use of pythons built in glob module.

This function handles recursive directory globing patterns by default.

Files are removed using `os.remove()`.

Parameters

- **glob_pattern** – The glob pattern to use to search for files to remove.
- **silent** – If True, don't print information to the tasks output.

Raises OSError – Raised if a file is in use (On Windows), or if there is another problem deleting one of the files.

glob_remove_dirs(*glob_pattern*, *silent=False*)

Remove directories using a glob pattern, this makes use of pythons built in glob module.

This function handles recursive directory globing patterns by default.

This uses `shutil.rmtree()` to remove directories.

This function will remove non empty directories.

Parameters

- **glob_pattern** – The glob pattern to use to search for directories to remove.
- **silent** – If True, don't print information to the tasks output.

makedirs(*path*, *mode=511*, *silent=False*, *exist_ok=True*)

Create a directory tree if it does not exist, if the directory tree exists already this function does nothing.

This uses `os.makedirs()`.

Parameters

- **path** – The directory path/tree.
- **mode** – The permissions umask to use for the directories.
- **silent** – If True, don't print information to the tasks output.
- **exist_ok** – If False, an OSError will be thrown if any directory in the given path already exists.

Raises OSError – Raised for all directory creation errors (aside from *errno.EEXIST* if **exist_ok** is **True**)

move(*self*, *src*, *dest*, *copy_function=shutil.copy2*, *silent=False*)

Recursively move a file or directory to another location. (See shutil.move) This is similar to the Unix "mv" command. Return the file or directory's destination.

If the destination is a directory or a symlink to a directory, the source is moved inside the directory. The destination path must not already exist.

If the destination already exists but is not a directory, it may be overwritten depending on `os.rename()` semantics.

If the destination is on our current filesystem, then `os.rename()` is used. Otherwise, src is copied to the destination and then removed. Symlinks are recreated under the new name if `os.rename()` fails because of cross filesystem renames.

The optional *copy_function* argument is a callable that will be used to copy the source or it will be delegated to `shutil.copytree()`. By default, `shutil.copy2()` is used, but any function that supports the same signature (like `shutil.copy()`) can be used.

Raises shutil.Error – If the destination already exists, or if src is moved into itself.

Parameters

- **src** – The file.

- **dest** – The destination to move the file to.
- **copy_function** – The copy function to use for copying individual files.
- **silent** – If True, don't print information to the tasks output.

printer

Return the printer object associated with this *pake.FileHelper*.

If one does not exist, return **None**.

remove (*path*, *silent=False*, *must_exist=False*)

Remove a file from disk if it exists, otherwise do nothing, uses `os.remove()`.

Raises

- **FileNotFoundError** – If must_exist is True, and the file does not exist.
- **OSError** – If the path is a directory.

Parameters

- **path** – The path of the file to remove.
- **silent** – If True, don't print information to the tasks output.
- **must_exist** – If set to True, a FileNotFoundError will be raised if the file does not exist.

rmtree (*path*, *silent=False*, *must_exist=False*)

Remove a directory tree if it exist, if the directory tree does not exists this function does nothing.

This uses `shutil.rmtree()`.

This function will remove non empty directories.

Raises **FileNotFoundError** – Raised if must_exist is True and the given path does not exist.

Parameters

- **path** – The directory path/tree.
- **silent** – If True, don't print information to the tasks output.
- **must_exist** – If True, a FileNotFoundError will be raised if the directory does not exist

touch (*file_name*, *mode=438*, *exist_ok=True*, *silent=False*)

Create a file at this given path. If mode is given, it is combined with the process' umask value to determine the file mode and access flags. If the file already exists, the function succeeds if exist_ok is true (and its modification time is updated to the current time), otherwise `FileExistsError` is raised.

This uses `pathlib.Path.touch()`.

Raises **FileExistsError** – Raised if **exist_ok** is **False** and the file already exists.

Parameters

- **file_name** – The file name.
- **mode** – The permissions umask.
- **exist_ok** – whether or not it is okay for the file to exist already.
- **silent** – If True, don't print information to the tasks output.

exception pake.**TaskException** (*task_name*, *exception*)

Bases: `Exception`

Raised by *pake.Pake.run()* if an exception is encountered running/visiting a task.

exception
The exception raised.

task_name
The name of the task which the exception was raised in.

exception_name
The fully qualified name of the exception object.

print_traceback(*file=None*)
Print the traceback of the exception that was raised inside the task to a file object.

Parameters file – The file object to print to. Default value is `pake.conf.stderr` if **None** is specified.

exception `pake.`**TaskExitException**(*task_name*, *exception*)
Bases: `Exception`

Raised when `SystemExit` or an exception derived from it is thrown inside a task.

This is raised from `pake.Pake.run()` when **exit()**, `pake.terminate()`, or `pake.Pake.terminate()` is called inside of a task.

task_name
The name of the task in which **exit** was called.

exception
Reference to the `SystemExit` exception which caused this exception to be raised.

print_traceback(*file=None*)
Print the traceback of the `SystemExit` exception that was raised inside the task to a file object.

Parameters file – The file object to print to. Default value is `pake.conf.stderr` if **None** is specified.

return_code
The return code passed to **exit()** inside the task.

exception `pake.`**TaskSubprocessException**(*cmd*, *returncode*, *output=None*, *output_stream=None*, *message=None*)
Bases: `pake.process.StreamingSubprocessException`

Raised by default upon encountering a non-zero return code from a subprocess spawned by the `pake.TaskContext` object.

This exception can be raised from `pake.TaskContext.call()`, `pake.TaskContext.check_call()`, and `pake.TaskContext.check_output()`.

cmd
Executed command in list form.

returncode
Process returncode.

message
Optional message from the raising function, may be **None**

filename
Filename describing the file from which the process call was initiated. (might be None)

function_name
Function name describing the function which initiated the process call. (might be None)

line_number
Line Number describing the line where the process call was initiated. (might be None)

`output`
All output of the process (including **stderr**) as a bytes object if it is available, otherwise this property is **None**.

`output_stream`
All output of the process (including **stderr**) as a file object at **seek(0)** if it is available, otherwise this property is **None**.

If this property is not **None** and you call *`pake.TaskSubprocessException.write_info()`*, this property will become **None** because that method reads the stream and disposes of it.

The stream will be a text mode stream.

`write_info`(*file*)
Writes information about the subprocess exception to a file like object.

This is necessary over implementing in __str__, because the process output might be drawn from another file to prevent issues with huge amounts of process output.

Calling this method will cause *`pake.TaskSubprocessException.output_stream`* to become **None** if it already isn't.

Parameters **`file`** – The text mode file object to write the information to.

`exception` `pake.`**`InputNotFoundException`**(*task_name*, *output_name*)
Bases: `Exception`

Raised by *`pake.Pake.run()`* and *`pake.Pake.dry_run()`* if a task with inputs declared cannot find an input file/directory on disk.

`exception` `pake.`**`MissingOutputsException`**(*task_name*)
Bases: `Exception`

Raised by *`pake.Pake.run()`* and *`pake.Pake.dry_run()`* if a task declares input files without specifying any output files/directories.

`exception` `pake.`**`UndefinedTaskException`**(*task_name*)
Bases: `Exception`

Raised on attempted lookup/usage of an unregistered task function or task name.

`task_name`
The name of the referenced task.

`exception` `pake.`**`RedefinedTaskException`**(*task_name*)
Bases: `Exception`

Raised on registering a duplicate task.

`task_name`
The name of the redefined task.

`exception` `pake.`**`PakeUninitializedException`**
Bases: `Exception`

Thrown if a function is called which depends on *`pake.init()`* being called first.

`exception` `pake.`**`SubpakeException`**(*cmd*, *returncode*, *output=None*, *output_stream=None*, *message=None*)
Bases: *`pake.process.StreamingSubprocessException`*

Raised upon encountering a non-zero return code from a subpake invocation.

This exception is raised from both *`pake.subpake()`* and *`pake.TaskContext.subpake()`*.

cmd
Executed subpake command in list form.

returncode
Process returncode.

message
Optional message from the raising function, may be **None**

filename
Filename describing the file from which the process call was initiated. (might be None)

function_name
Function name describing the function which initiated the process call. (might be None)

line_number
Line Number describing the line where the process call was initiated. (might be None)

output
All output of the process (including **stderr**) as a bytes object if it is available, otherwise this property is **None**.

output_stream
All output of the process (including **stderr**) as a file object at **seek(0)** if it is available, otherwise this property is **None**.

If this property is not **None** and you call `pake.TaskSubprocessException.write_info()`, this property will become **None** because that method reads the stream and disposes of it.

The stream will be a text mode stream.

write_info(*file*)
Writes information about the subprocess exception to a file like object.

This is necessary over implementing in __str__, because the process output might be drawn from another file to prevent issues with huge amounts of process output.

Calling this method will cause `pake.TaskSubprocessException.output_stream` to become **None** if it already isn't.

Parameters **file** – The text mode file object to write the information to.

exception pake.**TerminateException**(**args*)
Bases: `SystemExit`

This exception is raised by `pake.terminate()` and `pake.Pake.terminate()`, it derives `SystemExit` and if it is not caught pake will exit gracefully with the return code provided to the exception.

If this exception is raised inside of a task, `pake.Pake.run()` with raise a `pake.TaskExitException` in response.

code
exception code

exception pake.**AggregateException**(*exceptions*)
Bases: `Exception`

Thrown upon `pake.MultitaskContext.shutdown()` if the context had its `pake.MultitaskContext.aggregate_exceptions` setting set to **True** and one or more submitted tasks encountered an exception.

See the **aggregate_exceptions** parameter of `pake.TaskContext.multitask()`.

write_info(*file=None*)

Write information about all the encountered exceptions to a file like object. If you specify the file as **None**, the default is `pake.conf.stderr`

The information written is not guaranteed to be available for writing more than once.

Exceptions derived from *pake.process.StreamingSubprocessException* have special handling in this function, as they can incrementally stream information from a temp file and then dispose of it. They must have **write_info** called on them as well.

Parameters **file** – A text mode file like object to write information to.

2.2 Submodules

2.2.1 Module: pake.graph

class pake.graph.**Graph**

Bases: `object`

Represents a node in a directed graph.

add_edge(*edge*)

Add an edge to the graph.

Parameters **edge** – The edge to add (another *pake.graph.Graph* object)

edges

Retrieve a set of edges from this graph node.

Returns A set() of adjacent nodes.

remove_edge(*edge*)

Remove an edge from the graph by reference.

Parameters **edge** – Reference to a *pake.graph.Graph* object.

topological_sort()

Return a generator object that runs topological sort as it is iterated over.

Nodes that have been visited will not be revisited, making infinite recursion impossible.

Returns A generator that produces *pake.graph.Graph* nodes.

2.2.2 Module: pake.process

Methods for spawning processes.

pake.process.**DEVNULL**

Analog for `subprocess.DEVNULL`

pake.process.**STDOUT**

Analog for `subprocess.STDOUT`

pake.process.**PIPE**

Analog for `subprocess.PIPE`

exception pake.process.**ProcessException**(*message*)

Bases: `Exception`

Base class for process exceptions.

exception pake.process.**StreamingSubprocessException**(*cmd*, *returncode*, *output=None*, *output_stream=None*, *message=None*)

Bases: `pake.process.ProcessException`

A base class for sub-process exceptions which need to be able to handle reporting huge amounts of process output when a process fails.

This exception is used as a base class for process exceptions thrown from `pake.subpake()`, and the process spawning methods in the `pake.TaskContext` object.

cmd
Executed command in list form.

returncode
Process returncode.

message
Optional message from the raising function, may be **None**

filename
Filename describing the file from which the process call was initiated. (might be None)

function_name
Function name describing the function which initiated the process call. (might be None)

line_number
Line Number describing the line where the process call was initiated. (might be None)

output
All output of the process (including **stderr**) as a bytes object if it is available, otherwise this property is **None**.

output_stream
All output of the process (including **stderr**) as a file object at **seek(0)** if it is available, otherwise this property is **None**.

If this property is not **None** and you call `pake.TaskSubprocessException.write_info()`, this property will become **None** because that method reads the stream and disposes of it.

The stream will be a text mode stream.

write_info(*file*)
Writes information about the subprocess exception to a file like object.

This is necessary over implementing in __str__, because the process output might be drawn from another file to prevent issues with huge amounts of process output.

Calling this method will cause `pake.TaskSubprocessException.output_stream` to become **None** if it already isn't.

Parameters **file** – The text mode file object to write the information to.

exception pake.process.**CalledProcessException**(*cmd*, *returncode*, *output=None*, *stderr=None*)

Bases: `pake.process.ProcessException`

Raised when `pake.process.check_call()` or `pake.process.check_output()` and the process returns a non-zero exit status.

This exception is only raised by process spawning methods in the `pake.process` module.

cmd
Executed command

timeout
Timeout in seconds.

output
Output of the child process if it was captured by `pake.process.check_output()`. Otherwise, None.

stdout
Alias for output, for symmetry with stderr.

stderr
Stderr output of the child process if it was captured by `pake.process.check_output()`. Otherwise, None.

filename
Filename describing the file from which the process call was initiated. (might be None)

function_name
Function name describing the function which initiated the process call. (might be None)

line_number
Line Number describing the line where the process call was initiated. (might be None)

stdout
Alias for **output**

exception pake.process.**TimeoutExpired**(*cmd*, *timeout*, *output=None*, *stderr=None*)
Bases: `pake.process.ProcessException`

This exception is raised when the timeout expires while waiting for a child process.

This exception is only raised by process spawning methods in the `pake.process` module.

cmd
Executed command

timeout
Timeout in seconds.

output
Output of the child process if it was captured by `pake.process.check_output()`. Otherwise, None.

stdout
Alias for output, for symmetry with stderr.

stderr
Stderr output of the child process if it was captured by `pake.process.check_output()`. Otherwise, None.

filename
Filename describing the file from which the process call was initiated. (might be None)

function_name
Function name describing the function which initiated the process call. (might be None)

line_number
Line Number describing the line where the process call was initiated. (might be None)

pake.process.**call**(**args*, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *timeout=None*, ***kwargs*)
Wrapper around `subprocess.call()` which allows the same ***args** call syntax as `pake.TaskContext.call()` and friends.

Parameters

- **args** – Executable and arguments.
- **stdin** – Stdin to feed to the process.
- **stdout** – File to write stdout to.
- **stderr** – File to write stderr to.
- **shell** – Execute in shell mode.
- **timeout** – Program execution timeout value in seconds.

Raises *pake.process.TimeoutExpired* If the process does not exit before timeout is up.

`pake.process.`**`check_call`**(**args, stdin=None, stdout=None, stderr=None, shell=False, timeout=None, **kwargs*)

Wrapper around `subprocess.check_call()` which allows the same ***args** call syntax as *pake.TaskContext.call()* and friends.

Parameters

- **args** – Executable and arguments.
- **stdin** – Stdin to feed to the process.
- **stdout** – File to write stdout to.
- **stderr** – File to write stderr to.
- **shell** – Execute in shell mode.
- **timeout** – Program execution timeout value in seconds.

Raises *pake.process.CalledProcessException* If the process exits with a non-zero return code.

Raises *pake.process.TimeoutExpired* If the process does not exit before timeout is up.

`pake.process.`**`check_output`**(**args, stdin=None, stderr=None, shell=False, timeout=None, **kwargs*)

Wrapper around `subprocess.check_output()` which allows the same ***args** call syntax as *pake.TaskContext.call()* and friends.

Parameters

- **args** – Executable and arguments.
- **stdin** – Stdin to feed to the process.
- **stderr** – File to write stderr to.
- **shell** – Execute in shell mode.
- **timeout** – Program execution timeout value in seconds.

Raises *pake.process.CalledProcessException* If the process exits with a non-zero return code.

Raises *pake.process.TimeoutExpired* If the process does not exit before timeout is up.

2.2.3 Module: pake.util

`pake.util.`**`touch`**(*file_name, mode=438, exist_ok=True*)

Create a file at this given path. If mode is given, it is combined with the process' umask value to determine the file mode and access flags. If the file already exists, the function succeeds if **exist_ok** is true (and its modification time is updated to the current time), otherwise `FileExistsError` is raised.

Parameters

- **`file_name`** – The file name.
- **`mode`** – The mode (octal perms mask) defaults to **0o666**.
- **`exist_ok`** – Whether or not it is okay for the file to exist when touched, if not a `FileExistsError` is thrown.

`pake.util.`**`is_iterable`**`(`*obj*`)`

Test if an object is iterable.

Parameters **`obj`** – The object to test.

Returns True if the object is iterable, False otherwise.

`pake.util.`**`is_iterable_not_str`**`(`*obj*`)`

Test if an object is iterable, and not a string.

Parameters **`obj`** – The object to test.

Returns True if the object is an iterable non string, False otherwise.

`pake.util.`**`str_is_float`**`(`*value*`)`

Test if a string can be parsed into a float.

Returns True or False

`pake.util.`**`str_is_int`**`(`*value*`)`

Test if a string can be parsed into an integer.

Returns True or False

`pake.util.`**`flatten_non_str`**`(`*iterable*`)`

Flatten a nested iterable without affecting strings.

Example:

```
val = list(flatten_non_str(['this', ['is', ['an'], 'example']]))

# val == ['this', 'is', 'an', 'example']
```

Returns A generator that iterates over the flattened iterable.

`pake.util.`**`handle_shell_args`**`(`*args*`)`

Handles parsing the ***args** parameter of *`pake.TaskContext.call()`* and *`pake.subpake()`*.

It allows shell arguments to be passed as a list object, variadic parameters, or a single string.

Any non-string object you pass that is not an iterable will be stringified.

Returns A list of shell argument strings.

`class` `pake.util.`**`CallerDetail`**

`filename`

Source file name.

`function_name`

Function call name.

line_number
Line number of function call.

pake.util.**get_pakefile_caller_detail**()
Get the full pakefile path, called function name, and line number of the first function call in the current call tree which exists inside of a pakefile.

This function traverses up the stack frame looking for the first occurrence of a source file with the same path that *pake.get_init_file()* returns.

If *pake.init()* has not been called, this function returns **None**.

Returns A named tuple: *pake.util.CallerDetail* or **None**.

pake.util.**parse_define_value**(*value*)
Used to interpret the value of a define declared on the command line with the **-D/–define** option.

-D excepts all forms of python literal as define values.

This function can parse strings, integers, floats, lists, tuples, dictionaries and sets.

'True', 'False' and 'None' values are case insensitive.

Anything that does not start with a python literal quoting character (such as **{** and **[**) and is not a True or False value, Integer, or Float, is considered to be a raw string.

Raises ValueError if the **value** parameter is **None**, or if an attempt to parse a complex literal (quoted string, list, set, tuple, or dictionary) fails.

Parameters **value** – String representing the defines value.

Returns Python literal representing the defines values.

pake.util.**copyfileobj_tee**(*fsrc*, *destinations*, *length=16384*, *readline=False*)
copy data from file-like object **fsrc** to multiple file like objects.

Parameters

- **fsrc** – Source file object.
- **destinations** – List of destination file objects.
- **length** – Read chunk size, default is 16384 bytes.
- **readline** – If **True** readline will be used to read from **fsrc**, the **length** parameter will be ignored.

pake.util.**qualified_name**(*object_instance*)
Return the fully qualified type name of an object.

Parameters **object_instance** – Object instance.

Returns Fully qualified name string.

2.2.4 Module: pake.conf

Global configuration module.

pake.conf.**stdout**
(set-able) Default file object used by pake library calls to print informational output, defaults to **sys.stdout**

This can be set in order to change the default informational output location for the whole library.

`pake.conf.`**`stderr`**
(set-able) Default file object used by pake library calls to print error output, defaults to **sys.stderr**.

This can be set in order to change the default error output location for the whole library.

`pake.conf.`**`reset`**`()`
Reset all configuration to its default state.

2.2.5 Module: pake.returncodes

Pake return codes.

`pake.returncodes.`**`SUCCESS`**

0. Pake ran/exited successfully.

`pake.returncodes.`**`ERROR`**

1. Generic error, good for use with *`pake.terminate()`* (or **exit()** inside tasks)

`pake.returncodes.`**`PAKEFILE_NOT_FOUND`**

2. Pakefile not found in directory, or specified pakefile does not exist.

`pake.returncodes.`**`BAD_ARGUMENTS`**

3. Bad combination of command line arguments, or bad arguments in general.

`pake.returncodes.`**`BAD_DEFINE_VALUE`**

4. Syntax error while parsing a define value from the **-D/–define** option.

`pake.returncodes.`**`NO_TASKS_DEFINED`**

5. No tasks defined in pakefile.

`pake.returncodes.`**`NO_TASKS_SPECIFIED`**

6. No tasks specified to run, no default tasks exist.

`pake.returncodes.`**`TASK_INPUT_NOT_FOUND`**

7. One of task’s input files/directories is missing.

`pake.returncodes.`**`TASK_OUTPUT_MISSING`**

8. A task declares input files/directories but no output files/directories.

`pake.returncodes.`**`UNDEFINED_TASK`**

9. An undefined task was referenced.

`pake.returncodes.`**`TASK_SUBPROCESS_EXCEPTION`**

10. An unhandled *`pake.TaskSubprocessException`* was raised inside a task.

`pake.returncodes.`**`SUBPAKE_EXCEPTION`**

11. An exceptional condition occurred running a subpake script.

Or if a pakefile invoked with *`pake.subpake()`* returns non-zero and the subpake parameter **exit_on_error** is set to **True**.

`pake.returncodes.`**`TASK_EXCEPTION`**

12. An unhandled exception occurred inside a task.

`pake.returncodes.`**`AGGREGATE_EXCEPTION`**

13. An aggregate exception was raised from a usage of *pake.TaskContext.multitask()* where the **aggregate_exceptions** parameter of *pake.TaskContext.multitask()* was set to **True**.

pake.returncodes.**STDIN_DEFINES_SYNTAX_ERROR**

14. A syntax error was encountered parsing the defines dictionary passed into **stdin** while using the **–stdin-defines** option.

CHAPTER 3

Guides / Help

3.1 Running Pake

```
cd your_pakefile_directory

# Run pake with up to 10 tasks running in parallel

pake -j 10
```

pake will look for “pakefile.py” or “pakefile” in the current directory and run it if it exists.

3.1.1 Manually specifying pakefile(s)

You can specify one or more files to run with **-f/–file**. The switch does not have multiple arguments, but it can be used more than once to specify multiple files.

If you specify more than one pakefile with a **–jobs** parameter greater than 1, the specified pakefiles will still be run synchronously (one after another). The tasks inside each pakefile will be ran in parallel however.

For example:

```
pake -f pakefile.py foo

pake -f your_pakefile_1.py -f your_pakefile_2.py foo
```

3.1.2 Executing in another directory

The **-C** or **–directory** option can be used to execute pake in an arbitrary directory.

If you do not specify a file with **-f** or **–file**, then a pakefile must exist in the given directory:

Example:

```
# Pake will find the 'pakefile.py' in 'build_directory'
# then change directories into it and start running

pake -C build_directory my_task
```

You can also tell pake to run a pakefile (or multiple pakefiles) in a different working directory.

Example:

```
# Pake will run 'my_pakefile.py' with a working directory of 'build_directory'

pake -f my_pakefile.py -C build_directory my_task

# Pake will run all the given pakefiles with a working directory of 'build_directory'

pake -f pakefile1.py -f pakefile2.py -f pakefile3.py -C build_directory my_task
```

3.1.3 Running multiple tasks

You can specify multiple tasks, but do not rely on unrelated tasks being executed in any specific order because they won't be. If there is a specific order you need your tasks to execute in, the one that comes first should be declared a dependency of the one that comes second, then the second task should be specified to run.

When running parallel builds, leaf dependencies will start executing pretty much simultaneously, and non related tasks that have a dependency chain may execute in parallel.

In general, direct dependencies of a task have no defined order of execution when there is more than one of them.

```
pake task unrelated_task order_independent_task
```

3.1.4 Specifying define values

The **-D/–define** option is used to specify defines on the command line that can be retrieved with the *pake.Pake.get_define()* method, or **__getitem__** indexer on the *pake.Pake* object (which is returned by *pake.init()*).

Define values are parsed partially with the built in `ast` module, the only caveat is that the values **True**, **False** and **None** are case insensitive.

Defines which are specified without a value, default to the value of **True**.

Basic Example:

```
pake -D IM_TRUE=True \
     -D IM_TRUE_TOO=true \
     -D IM_NONE=none \
     -D NO_VALUE \
     -D IM_STRING="Hello" \
     -D IM_INT=1 \
     -D IM_FLOAT=0.5
```

Retrieval:

```
import pake

pk = pake.init()

```

(continues on next page)

(continued from previous page)

```
im_true = pk.get_define('IM_TRUE')

im_true_too = pk.get_define('IM_TRUE_TOO')

im_none = pk.get_define('IM_NONE')

no_value = pk.get_define('NO_VALUE')

im_string = pk.get_define('IM_STRING')

im_int = pk.get_define('IM_INT')

im_float = pk.get_define('IM_FLOAT')


print(type(im_true)) # -> <class 'bool'>
print(im_true) # -> True

print(type(im_true_too)) # -> <class 'bool'>
print(im_true_too) # -> True

print(type(im_none)) # -> <class 'NoneType'>
print(im_none) # -> None

print(type(no_value)) # -> <class 'bool'>
print(no_value) # -> True

print(type(im_string)) # -> <class 'str'>
print(im_string) # -> Hello

print(type(im_int)) # -> <class 'int'>
print(im_int) # -> 1

print(type(im_float)) # -> <class 'float'>
print(im_float) # -> 0.5

pk.terminate(0)
```

You can pass complex python literals such as lists, sets, tuples, dictionaries, etc.. as a define value. pake will recognize and fully deserialize them into the correct type.

Complex Types Example:

```
pake -D IM_A_DICT="{'im': 'dict'}" \
     -D IM_A_SET="{'im', 'set'}" \
     -D IM_A_LIST="['im', 'list']" \
     -D IM_A_TUPLE="('im', 'tuple')"
```

Retrieval:

```
import pake

pk = pake.init()

im_a_dict = pk.get_define('IM_A_DICT')

im_a_set = pk.get_define('IM_A_SET')
```

(continues on next page)

(continued from previous page)

```
im_a_list = pk.get_define('IM_A_LIST')

im_a_tuple = pk.get_define('IM_A_TUPLE')


print(type(im_a_dict)) # -> <class 'dict'>
print(im_a_dict) # -> {'im': 'dict'}

print(type(im_a_set)) # -> <class 'set'>
print(im_a_set) # -> {'im', 'set'}

print(type(im_a_list)) # -> <class 'list'>
print(im_a_list) # -> ['im': 'list']

print(type(im_a_tuple)) # -> <class 'tuple'>
print(im_a_tuple) # -> ('im': 'tuple')

pk.terminate(0)
```

3.1.5 Reading defines from STDIN

The **–stdin-defines** option allows you to pipe defines into pake in the form of a python dictionary.

Any defines that are set this way can be overwritten by defines set on the command line using **-D/–define**

The dictionary that you pipe in is parsed into a python literal using the built in `ast` module, so you can use complex types such as lists, sets, tuples, dictionaries ect.. as the value for your defines.

Pake reads the defines from **stdin** on the first call to *pake.init()* and caches them in memory. Later calls to **init** will read the specified defines back from cache and apply them to a newly created *pake.Pake* instance.

Calls to *pake.de_init()* will not clear cached defines read from **stdin**.

Example Pakefile:

```
import pake

pk = pake.init()

a = pk['MY_DEFINE']
b = pk['MY_DEFINE_2']

print(a)
print(b)

pk.terminate(0)
```

Example Commands:

```
# Pipe in two defines, MY_DEFINE=True and MY_DEFINE_2=42

echo "{'MY_DEFINE': True, 'MY_DEFINE_2': 42}" | pake --stdin-defines

# Prints:
```

(continues on next page)

(continued from previous page)

```
True
42


# Overwrite the value of MY_DEFINE_2 that was piped in, using the -D/--define option
# it will have a value of False instead of 42

echo "{'MY_DEFINE': True, 'MY_DEFINE_2': 42}" | pake --stdin-defines -D MY_DEFINE_
↪2=False

# Prints:

True
False
```

3.1.6 Environmental variables

Pake currently recognizes only one environmental variable named `PAKE_SYNC_OUTPUT`.

This variable corresponds to the command line option **–sync-output**. Using the **–sync-output** option will override the environmental variable however. Pake will use the value from the command line option instead of the value in the environment.

When this environmental variable and **–sync-output** are not defined/specified, the default value pake uses is **–sync-output True**.

`pake.init()` has an argument named **sync_output** that can also be used to permanently override both the **–sync-output** switch and the `PAKE_SYNC_OUTPUT` environmental variable from inside of a pakefile.

The **–sync-output** option controls whether pake tries to synchronize task output by queueing it when running with more than one job.

–sync-output False causes `pake.TaskContext.io_lock` to yield a lock object which actually does nothing when it is acquired, and it also forces pake to write all run output to `pake.Pake.stdout` instead of task output queues, even when running tasks concurrently.

The output synchronization setting is inherited by all `pake.subpake()` and `pake.Pake.subpake()` invocations.

You can stop this inheritance by manually passing a different value for **–sync-output** as a shell argument when using one of the **subpake** functions. The new value will be inherited by the pakefile you invoked with **subpake** and all of its children.

3.1.7 Command line options

```
usage: pake [-h] [-v] [-D DEFINE] [--stdin-defines] [-j JOBS] [-n]
                [-C DIRECTORY] [-t] [-ti] [--sync-output {True, False, 1, 0}]
                [-f FILE]
                [tasks [tasks ...]]

    positional arguments:
      tasks                 Build tasks.

    optional arguments:
      -h, --help            show this help message and exit
```

(continues on next page)

(continued from previous page)

```
  -v, --version         show program's version number and exit
  -D DEFINE, --define DEFINE
                        Add defined value.
  --stdin-defines       Read defines from a Python Dictionary piped into
                        stdin. Defines read with this option can be
                        overwritten by defines specified on the command line
                        with -D/--define.
  -j JOBS, --jobs JOBS  Max number of parallel jobs. Using this option enables
                        unrelated tasks to run in parallel with a max of N
                        tasks running at a time.
  -n, --dry-run         Use to preform a dry run, lists all tasks that will be
                        executed in the next actual invocation.
  -C DIRECTORY, --directory DIRECTORY
                        Change directory before executing.
  -t, --show-tasks      List all task names.
  -ti, --show-task-info
                        List all tasks along side their doc string. Only tasks
                        with doc strings present will be shown.
  --sync-output {True, False, 1, 0}
                        Tell pake whether it should synchronize task output
                        when running with multiple jobs. Console output can
                        get scrambled under the right circumstances with this
                        turned off, but pake will run slightly faster. This
                        option will override any value in the PAKE_SYNC_OUTPUT
                        environmental variable, and is inherited by subpake
                        invocations unless the argument is re-passed with a
                        different value or overridden in pake.init.
  -f FILE, --file FILE  Pakefile path(s). This switch can be used more than
                        once, all specified pakefiles will be executed in
                        order with the current directory as the working
                        directory (unless -C is specified).
```

3.1.8 Return codes

See the *pake.returncodes* module, pake's return codes are defined as constants and each is described in detail in the module documentation.

3.2 Writing Basic Tasks

Additional information about change detection is available in the form of examples in the documentation for the *pake.Pake.task()* function decorator.

Pake is capable of handling change detection against both files and directories, and the two can be used as inputs or outputs interchangeably and in combination.

Note:

Each registered task receives a *pake.TaskContext* instance as a single argument when run. In this example the argument is named **ctx**, but it can be named however you like. It is not an error to leave this argument undefined, but you will most likely be using it.

Example:

```
import pake

# Tasks are registered the the pake.Pake object
# returned by pake's initialization call, using the task decorator.

pk = pake.init()

# Try to grab a command line define.
# In particular the value of -D CC=..
# CC will default to 'gcc' in this case if
# it was not specified.

CC = pk.get_define('CC', 'gcc')

# you can also use the syntax: pk["CC"] to
# attempt to get the defines value, if it is not
# defined then it will return None.

# ===

# If you just have a single input/output, there is no
# need to pass a list to the tasks inputs/outputs

@pk.task(i='foo/foo.c', o='foo/foo.o')
def foo(ctx):
    # Execute a program (gcc) and print its stdout/stderr to the tasks output.

    # ctx.call can be passed a command line as variadic arguments, an iterable, or
    # as a string.  It will automatically flatten out non string iterables in your␣
↪variadic
    # arguments or iterable object, so you can pass an iterable such as ctx.inputs
    # as part of your full command line invocation instead of trying to create the␣
↪command
    # line by concatenating lists or using the indexer on ctx.inputs/ctx.outputs

    ctx.call(CC, '-c', ctx.inputs, '-o', ctx.outputs)


# Pake can handle file change detection with multiple inputs
# and outputs. If the amount of inputs is different from
# the amount of outputs, the task is considered to be out
# of date if any input file is newer than any output file.

# When the amount of inputs is equal to the amount of outputs,
# pake will compare each input to its corresponding output
# and collect out of date input/outputs into ctx.outdated_inputs
# and ctx.outdated_outputs respectively.  ctx.outdated_pairs
# can be used to get a generator over (input, output) pairs,
# it is shorthand for zip(ctx.outdated_inputs, ctx.outdated_outputs)

@pk.task(i=pake.glob('bar/*.c'), o=pake.pattern('bar/%.o'))
def bar(ctx):

    # zip together the outdated inputs and outputs, since they
    # correspond to each other, this iterates of a sequence of python
    # tuple objects in the form (input, output)
```

(continues on next page)

(continued from previous page)

```
    for i, o in ctx.outdated_pairs:
        ctx.call(CC, '-c', i, '-o', o)


# This task depends on the 'foo' and 'bar' tasks, as
# specified with the decorators leading parameters.
# It outputs 'bin/baz' by taking the input 'main.c'
# and linking it to the object files produced in the other tasks.

@pk.task(foo, bar, o='bin/baz', i='main.c')
def baz(ctx):
    """Use this to build baz"""

    # Documentation strings can be viewed by running 'pake -ti' in
    # the directory the pakefile exists in, it will list all documented
    # tasks with their python doc strings.

    # The pake.FileHelper class can be used to preform basic file
    # system operations while printing information about the operations
    # it has completed to the tasks output.

    file_helper = pake.FileHelper(ctx)

    # Create a bin directory, this won't complain if it exists already
    file_helper.makedirs('bin')

    # ctx.dependency_outputs contains a list of all outputs that this
    # tasks immediate dependencies produce

    ctx.call(CC, '-o', ctx.outputs, ctx.inputs, ctx.dependency_outputs)


@pk.task
def clean(ctx):
    """Clean binaries"""

    file_helper = pake.FileHelper(ctx)

    # Clean up using the FileHelper object.
    # Remove the bin directory, this wont complain if 'bin'
    # does not exist.

    file_helper.rmtree('bin')

    # Glob remove object files from the foo and bar directories

    file_helper.glob_remove('foo/*.o')
    file_helper.glob_remove('bar/*.o')


# Run pake; The default task that will be executed when
# none are specified on the command line will be 'baz' in
# this case.

# The tasks parameter is optional, but if it is not specified
# here, you will be required to specify a task or tasks on the
# command line.
```

(continues on next page)

(continued from previous page)

```
pake.run(pk, tasks=baz)
```

Output from command `pake`:

```
===== Executing task: "bar"
gcc -c bar/bar.c -o bar/bar.o
===== Executing task: "foo"
gcc -c foo/foo.c -o foo/foo.o
===== Executing task: "baz"
Created Directory(s): "bin"
gcc -o bin/baz main.c foo/foo.o bar/bar.o
```

Output from command `pake clean`:

```
===== Executing task: "clean"
Removed Directory(s): "bin"
Glob Removed Files: "foo/*.o"
Glob Removed Files: "bar/*.o"
```

3.3 Input/Output Name Generators & Globbing

Pake can accept callables as task inputs, this is how *pake.glob()* and *pake.pattern()* are implemented under the hood. They both take the pattern you give them and return a new function, which generates an iterable of input/output file names.

Input and output generators will work with both the *pake.Pake.task()* function decorator, as well as the *pake.Pake.add_task()* method.

The evaluation of input/output generators is deferred until the task runs.

The reasoning behind this is that input generators like *pake.glob()* can have access to artifacts created by a tasks dependencies when evaluation happens just before the task runs. Because all the dependencies of the task will have been built by that point.

The outputs of input/output generators are passed through pake's change detection algorithm once the values have been retrieved from them, to determine if the task should run or not.

pake.glob() and *pake.pattern()* are implemented like this:

```
def glob(pattern):

    def input_generator():

        # Input generators can return generators or lists.
        # As long as whatever the input generator returns
        # is an iterable object that produces file/directory
        # names as strings, it will work.

        # You can also use 'yield' syntax in your input
        # generator function, since the result is an iterable

        return glob.iglob(expression, recursive=True)

    return input_generator
```

(continues on next page)

(continued from previous page)

```python
# pake.pattern generates a callable that you can
# pass as a pake output, into the "i" parameter
# of the task decorator, or even to the "inputs"
# parameter of pk.add_task

def pattern(file_pattern):

    def output_generator(inputs):

        # output generators receive the inputs
        # provided to the task, even ones that
        # were generated by an input generator

        # inputs is always a flat list, and a copy

        # inputs is safe to mutate if you want

        # As with input generators, you can return
        # any type of iterable that produces file/directory
        # names as strings and it will work.

        for inp in inputs:
            dir = os.path.dirname(inp)
            name, ext = os.path.splitext(os.path.basename(inp))
            yield file_pattern.replace('{dir}', dir).replace('%', name).replace('{ext}
↪', ext)

    return output_generator
```

pake.glob() and *pake.pattern()* are used like this:

```python
import pake

pk = pake.init()

@pk.task(i=pake.glob('src/*.c'), o=pake.pattern('bin/%.o'))
def build_c(ctx):

    # Your going to have an equal number of
    # inputs and outputs in this task, because
    # the output names are being generated from
    # the input names.

    for i, o in ctx.outdated_pairs:
        ctx.call('gcc', '-c', i, '-o', o)

pake.run(pk, tasks=build_c)
```

3.3.1 Multiple input generators at once

You can place input generators into a list or any other iterable such as a tuple, pake will combine the values that they generate into one flat list. You can also use input generator callables along side plain old file or directory references.

Multiple output generators are not allowed however, you may only ever use one output generator callable at a time, and you cannot use output generators along side regular file/directory names.

Output generators process all of the tasks input file/directory names, and are expected to return all of the tasks outputs.

Multiple Input Generator Example:

```
import pake

pk = pake.init()

# This task collects .c files from two directories
# ('src_one' and 'src_two'), and compiles them all
# together with 'main.c' (which exists in the current directory).

# This task produces an executable file called 'main'

@pk.task(i=[pake.glob('src_one/*.c'), pake.glob('src_two/*.c'), 'main.c'], o='main')
def build_c(ctx):
    ctx.call('gcc', ctx.inputs, '-o', ctx.outputs)


pake.run(pk, tasks=build_c)
```

Example with an output generator:

```
import pake

pk = pake.init()

# This task collects .c files from two directories
# ('src_one' and 'src_two'), and compiles object files
# that are created in each source directory along side
# the source file.

@pk.task(i=[pake.glob('src_one/*.c'), pake.glob('src_two/*.c')], o=pake.pattern('{dir}
↪/%.o'))
def build_c(ctx):

    # Your going to have an equal number of
    # inputs and outputs in this task, because
    # the output names are being generated from
    # the input names.

    for i, o in ctx.outdated_pairs:
        ctx.call('gcc', '-c', i, '-o', o)


pake.run(pk, tasks=build_c)
```

3.4 Change Detection Against Directories

Change detection in pake works against directories in the same way it works against files.

Files can be compared against directories (and vice versa) when providing inputs and outputs to a task, directories can also be compared against each other if needed.

Basically, a directory name can be used in place of a file name anywhere in a tasks input(s) and output(s) parameters.

Example:

```
import pake
import glob
import pathlib

pk = pake.init()

# Whenever the modification time of 'my_directory' or
# 'my_directory_2' is more recent than the file 'my_big.png',
# this task will run.

@pk.task(i=['my_directory', 'my_directory_2'], o='my_big.png')
def concatenate_pngs(ctx):
   png_files = []

   for d in ctx.inputs:
       # Need to collect the files in the directories yourself
       png_files += pathlib.Path(d).glob('*.png')

   # Concatenate with ImageMagick's convert command
   ctx.call('convert', png_files, '-append', ctx.outputs)


pake.run(pk, tasks=concatenate_pngs)
```

3.5 Exiting Pakefiles Gracefully

`pake.terminate()` can be used to gracefully exit a pakefile from anywhere.

You can also use `pake.Pake.terminate()` on the pake context returned by `pake.init()`.

`pake.Pake.terminate()` is just a shortcut for calling `pake.terminate()` with the first argument filled out.

These methods are for exiting pake with a given return code after it is initialized, they ensure the proper 'leaving directory / exit subpake' messages are sent to pake's output if needed upon exit, and help keep logged output consistent.

You should use these functions instead of **exit** when handling error conditions that occur outside of pake tasks before `pake.run()` is called.

It is optional to use `pake.terminate()` inside tasks, **exit** will always work inside tasks but `pake.terminate()` may provide additional functionality in the future.

Example Use Case:

```
import os
import pake
from pake import returncodes

pk = pake.init()

# Say you need to wimp out of a build for some reason
# But not inside of a task.  pake.terminate will make sure the
# 'leaving directory/exiting subpake' message is printed
# if it needs to be.

if os.name == 'nt':
```

(continues on next page)

(continued from previous page)

```
    pk.print('You really thought you could '
             'build my software on windows? nope!')

    pake.terminate(pk, returncodes.ERROR)

    # or

    # pk.terminate(returncodes.ERROR)


# Define some tasks...

@pk.task
def build(ctx):
    # You can use pake.terminate() inside of a task as well as exit()
    # pake.terminate() may offer more functionality than a raw exit()
    # in the future, however exit() will always work too.

    something_bad_happened = True

    if something_bad_happened:
        pake.terminate(pk, returncodes.ERROR)

        # Or:

        # pk.terminate(returncodes.ERROR)

pake.run(pk, tasks=build)

# If you were to use pk.run, a TaskExitException would be thrown
# the inner exception (err.exception) would be set to
# pake.TerminateException

# try:
#     pk.run(tasks=test)
# except pake.TaskExitException as err:
#     print('\n' + str(err) + '\n')
#
#     # print to pake.conf.stderr by default
#     # file parameter can be used to change that
#     err.print_traceback()
```

3.5.1 Calls To exit() inside tasks

You can also exit pake with a specific return code when inside a task by simply calling **exit**.

exit inside of a task is considered a global exit, even when a task is on another thread due to pake's **–jobs** parameter being greater than 1. The return code passed to **exit** inside the task will become the return code for command line call to pake.

exit will always work inside of a task and cause a graceful exit, however *pake.terminate()* may offer more functionality than **exit** sometime in the future.

If you exit with *pake.returncodes.SUCCESS*, no stack trace for the exit call will be printed.

Pake handles calls to **exit** in the same manner as it handles exceptions, although this condition is instead signified by a *pake.TaskExitException* from *pake.Pake.run()* and the message sent to pake's output is slightly different.

The behavior when running parallel pake is the same as when a normal exception is thrown.

Example:

```
import pake
from pake import returncodes

pk = pake.init()

@pk.task
def test(ctx):
    ctx.print('hello world')

    # We could also use anything other than 0 to signify an error.
    # returncodes.SUCCESS and returncodes.ERROR will always be 0 and 1.
    exit(returncodes.ERROR)

pake.run(pk, tasks=test)

# If you were to use pk.run, a TaskExitException would be thrown

# try:
#     pk.run(tasks=test)
# except pake.TaskExitException as err:
#     print('\n' + str(err) + '\n')
#
#     # print to pake.conf.stderr by default
#     # file parameter can be used to change that
#     err.print_traceback()
```

Yields Output:

```
===== Executing Task: "test"
hello world

Exit exception "SystemExit" with return-code(1) was raised in task "test".

Traceback (most recent call last):
  File "{PAKE_INSTALL_PATH}/pake/pake.py", line 1316, in func_wrapper
    return func(*args, **kwargs)
  File "{FULL_PAKEFILE_DIR_PATH}/pakefile.py", line 12, in test
    exit(returncodes.ERROR)
  File "{PYTHON_INSTALL_PATH}/lib/_sitebuiltins.py", line 26, in __call__
    raise SystemExit(code)
SystemExit: 1
```

3.5.2 Stack traces from exit/terminate in tasks

Calls to **exit()**, `pake.terminate()`, or `pake.Pake.terminate()` with non-zero return codes will result in a stack trace being printed with information about the location of the exit or terminate call.

This is not the case if you call **exit()** or pake's terminate functions with a return code of zero, there will be no stack trace or any information printed if the return code indicates success.

Example **exit(1)** stack trace:

```python
import pake
from pake import returncodes

pk = pake.init()


@pk.task
def build(ctx):
    exit(returncodes.ERROR)

pake.run(pk, tasks=build)
```

Yields Output:

```
===== Executing Task: "build"

Exit exception "SystemExit" with return-code(1) was raised in task "build".

Traceback (most recent call last):
  File "{PAKE_INSTALL_PATH}/pake/pake.py", line 1504, in func_wrapper
    return func(*args, **kwargs)
  File "{FULL_PAKEFILE_DIR_PATH}/pakefile.py", line 9, in build
    exit(returncodes.ERROR)
  File "{PYTHON_INSTALL_PATH}/lib/_sitebuiltins.py", line 26, in __call__
    raise SystemExit(code)
SystemExit: 1
```

Example **terminate(1)** stack trace:

```python
import pake
from pake import returncodes

pk = pake.init()


@pk.task
def build(ctx):
    pk.terminate(returncodes.ERROR)

pake.run(pk, tasks=build)
```

Yields Output:

```
===== Executing Task: "build"

Exit exception "pake.program.TerminateException" with return-code(1) was raised in␣
↪task "build".

Traceback (most recent call last):
  File "{PAKE_INSTALL_PATH}/pake/pake.py", line 1504, in func_wrapper
    return func(*args, **kwargs)
  File "{FULL_PAKEFILE_DIR_PATH}/pakefile.py", line 9, in build
    pk.terminate(returncodes.ERROR)
  File "{PAKE_INSTALL_PATH}/pake/pake.py", line 1027, in terminate
    pake.terminate(self, return_code=return_code)
  File "{PAKE_INSTALL_PATH}/pake/program.py", line 614, in terminate
    m_exit(return_code)
```

(continues on next page)

(continued from previous page)

```
  File "{PAKE_INSTALL_PATH}/pake/program.py", line 605, in m_exit
    raise TerminateException(code)
pake.program.TerminateException: 1
```

3.6 Adding Tasks Programmatically

Pake tasks may be programmatically added using the *pake.Pake.add_task()* method of the pake instance.

When adding tasks programmatically, you may specify a callable class instance or a function as your task entry point.

Basic C to Object Compilation Task Example:

```
import pake

pk = pake.init()


def compile_c(ctx):
   for i, o in ctx.outdated_pairs:
       ctx.call(['gcc', '-c', i, '-o', o])

# The task name may differ from the function name.

pk.add_task('compile_c_to_objects', compile_c,
           inputs=pake.glob('src/*.c'),
           outputs=pake.pattern('obj/%.o'))

pake.run(pk, tasks='compile_c_to_objects')

# Or:

# pake.run(pk, tasks=compile_c)
```

Multiple Dependencies:

```
import pake

pk = pake.init()

@pk.task
def task_a():
    pass

@pk.task
def task_b():
    pass

def do_both():
    pass

# The dependencies parameter will accept a single task reference
# as well as a list of task references

pk.add_task('do_both', do_both, dependencies=[task_a, 'task_b'])
```

(continues on next page)

(continued from previous page)

```
pake.run(pk, tasks=do_both)
```

Callable Class Example:

```
import pake

pk = pake.init()

class MessagePrinter:
    """Task documentation goes here."""

    def __init__(self, message):
        self._message = message

    def __call__(self, ctx):
        ctx.print(self._message)


pk.add_task('task_a', MessagePrinter('Hello World!'))

instance_a = MessagePrinter('hello world again')


# Can refer to the dependency by name, since we did not save a reference.

pk.add_task('task_b', instance_a, dependencies='task_a')


instance_b = MessagePrinter('Goodbye!')

# Can also refer to the dependency by instance.

pk.add_task('task_c', instance_b, dependencies=instance_a)

pake.run(pk, tasks='task_c')

# Or:

# pake.run(pk, tasks=instance_b)
```

3.7 Exceptions Inside Tasks

Pake handles most exceptions occuring inside a task by wrapping them in a *pake.TaskException* and throwing them from *pake.Pake.run()*.

pake.run() handles all of the exceptions from *pake.Pake.run()* and prints the exception information to `pake.conf.stderr` in a way that is useful to the user/developer.

Example:

```
import pake

pk = pake.init()
```

(continues on next page)

(continued from previous page)

```
@pk.task
def test(ctx):
    ctx.print('hello world')
    raise Exception('Some Exception')

pake.run(pk, tasks=test)

# If you were to use pk.run, a TaskException would be thrown

# try:
#     pk.run(tasks=test)
# except pake.TaskException as err:
#     print('\n'+str(err)+'\n')
#
#     # print to pake.conf.stderr by default
#     # file parameter can be used to change that
#     err.print_traceback()
```

Yields Output:

```
===== Executing Task: "test"
hello world

Exception "Exception" was raised within task "test".

Traceback (most recent call last):
  File "{PAKE_INSTALL_PATH}/pake/pake.py", line 1316, in func_wrapper
    return func(*args, **kwargs)
  File "{FULL_PAKEFILE_DIR_PATH}/pakefile.py", line 8, in test
    raise Exception('Some Exception')
Exception: Some Exception
```

When an exception is thrown inside a task, the fully qualified exception name and the task it occurred in will be mentioned at the very end of pake's output. That information is followed by a stack trace for the raised exception.

When running with multiple jobs, pake will stop as soon as possible. Independent tasks that were running in the background when the exception occurred will finish, and then the information for the encountered exception will be printed at the very end of pake's output.

3.7.1 pake.TaskSubprocessException

Special error reporting is implemented for *pake.TaskSubprocessException*, which is raised from *pake.TaskContext.call*, *pake.TaskContext.check_call*, and *pake.TaskContext.check_output*.

When a process called through one of these process spawning methods returns with a non-zero return code, a *pake.TaskSubprocessException* is raised by default. That will always be true unless you have supplied **ignore_errors=True** as an argument to these functions.

This exception derives from *pake.process.StreamingSubprocessException*, an exception base class which incrementally reads process output that has been buffered to disk when reporting error information. Buffering the output to disk and reading it back incrementally helps keep huge amounts of process output from crashing pake.

The reported exception information will contain the full path to your pakefile, the name of the process spawning function, and the line number where it was called. All of this will be at the very top of the error message.

All output from the failed command will be mentioned at the bottom in a block surrounded by brackets, which is labeled with **"Command Output: "**

Example:

```
import pake


pk = pake.init()

@pk.task
def test(ctx):
    # pake.TaskSubprocessException is raised because
    # which cannot find the given command and returns non-zero

    # silent is specified, which means the process will not
    # send any output to the task IO queue, but the command
    # will still be printed
    ctx.call('which', "i-dont-exist", silent=True)

pake.run(pk, tasks=test)
```

Yields Output:

```
===== Executing Task: "test"
which i-dont-exist

pake.pake.TaskSubprocessException(
        filename="{FULL_PAKEFILE_DIR_PATH}/pakefile.py",
        function_name="call",
        line_number=9
)

Message: A subprocess spawned by a task exited with a non-zero return code.

The following command exited with return code: 1

which i-dont-exist

Command Output: {

which: no i-dont-exist in ({EVERY_DIRECTORY_IN_YOUR_ENV_PATH_VAR})


}
```

3.7.2 pake.SubpakeException

pake.SubpakeException is derived from *pake.process.StreamingSubprocessException* just like *pake.TaskSubprocessException*, and produces similar error information when raised inside a task.

Example: `subfolder/pakefile.py`

```
import pake

pk = pake.init()

@pk.task
def sub_test(ctx):
    raise Exception('Test Exception')
```

(continues on next page)

(continued from previous page)

```
pake.run(pk, tasks=sub_test)
```

Example: `pakefile.py`

```python
import pake

pk = pake.init()

@pk.task
def test(ctx):
    # pake.SubpakeException is raised because
    # 'subfolder/pakefile.py' raises an exception inside a task
    # and returns with a non-zero exit code.

    # Silent prevents the pakefiles output from being printed
    # to the task IO queue, keeping the output short for this example

    ctx.subpake('subfolder/pakefile.py', silent=True)

pake.run(pk, tasks=test)
```

Yields Output:

```
===== Executing Task: "test"

pake.subpake.SubpakeException(
        filename="{REST_OF_FULL_PATH}/pakefile.py",
        function_name="subpake",
        line_number=13
)

Message: A pakefile invoked by pake.subpake exited with a non-zero return code.

The following command exited with return code: 13

{PYTHON_INSTALL_DIR}/python3 subfolder/pakefile.py --_subpake_depth 1 --stdin-defines␣
↪--directory {REST_OF_FULL_PATH}/subfolder

Command Output: {

*** enter subpake[1]:
pake[1]: Entering Directory "{REST_OF_FULL_PATH}/subfolder"
===== Executing Task: "sub_test"

Exception "Exception" was called within task "sub_test".

Traceback (most recent call last):
  File "{PAKE_INSTALL_DIRECTORY}/pake/pake.py", line 1323, in func_wrapper
    return func(*args, **kwargs)
  File "subfolder/pakefile.py", line 7, in sub_test
Exception: Test Exception

pake[1]: Exiting Directory "{REST_OF_FULL_PATH}/subfolder"
*** exit subpake[1]:
```

(continues on next page)

(continued from previous page)

```
}
```

3.8 Concurrency Inside Tasks

Work can be submitted to the threadpool pake is running its tasks on to achieve a predictable level of concurrency for sub tasks that is limited by the **–jobs** command line argument, or the **jobs** parameter of *pake.run()* and *pake.Pake.run()*.

This is done using the *pake.MultitaskContext* returned by *pake.TaskContext.multitask()*.

pake.MultitaskContext implements an **Executor** with an identical interface to `concurrent.futures.ThreadPoolExecutor` from the built-in python module `concurrent.futures`.

Submitting work to a *pake.MultitaskContext* causes your work to be added to the threadpool that pake is running on when the **–jobs** parameter is greater than **1**.

When the **–jobs** parameter is **1** (the default value), *pake.MultitaskContext* degrades to synchronous behavior.

Example:

```
import pake

# functools.partial is used for binding argument values to functions

from functools import partial


pk = pake.init()


@pk.task(i=pake.glob('src/*.c'), o=pake.pattern('obj/%.o'))
def build_c(ctx)

   file_helper = pake.FileHelper(ctx)

   # Make 'obj' directory if it does not exist.
   # This does not complain if it is already there.

   file_helper.makedirs('obj')

   # Start multitasking

   with ctx.multitask() as mt:
       for i, o in ctx.outdated_pairs:

           # Read the section 'Output synchronization with ctx.call & ctx.subpake'
           # near the bottom of this page for an explanation of 'sync_call'
           # below, and how output synchronization is achieved for
           # ctx.call and ctx.subpake

           sync_call = partial(ctx.call,
                               collect_output=pk.max_jobs > 1)

           # Submit a work function with arguments to the threadpool
           mt.submit(sync_call, ['gcc', '-c', i, '-o', o])
```

(continues on next page)

(continued from previous page)

```python
@pk.task(build_c, i=pake.glob('obj/*.o'), o='main')
def build(ctx):

   # Utilizing the automatic non string iterable
   # flattening here to pass ctx.inputs and ctx.outputs

   ctx.call('gcc', ctx.inputs, '-o', ctx.outputs)


pake.run(pk, tasks=build)
```

3.8.1 Output synchronization with ctx.print & ctx.io.write

If you are using *pake.TaskContext.multitask()* to add concurrency to the inside of a task, you are in charge of synchronizing output to the task IO queue.

Pake will synchronize writing the whole task IO queue when the task finishes if **–sync-output False** is not specified on the command line, but it will not be able to synchronize the output from sub tasks you submit to its threadpool by yourself without help.

When performing multiple writes to *pake.TaskContext.io()* from inside of a task submitted to *pake.MultitaskContext()*, you need to acquire a lock on *pake.TaskContext.io_lock* if you want to sure all your writes show up in the order you made them.

If **–sync-output False** is specified on the command line or *pake.Pake.sync_output* is set to **False** manually in the pakefile, then using *pake.TaskContext.io_lock* in a **with** statement does not actually acquire any lock.

If you know that the function or subprocess you are calling is only ever going to write **once** to the task IO queue (such as the functions in *pake.FileHelper*), then there is no need to synchronize the output. The single write may come out of order with respect to other sub tasks, but the message as a whole will be intact/in-order.

Example:

```python
import pake
import random
import time


pk = pake.init()


def my_sub_task(ctx):

    data = [
        'Hello ',
        'World, ',
        'I ',
        'Come ',
        'On ',
        'One ',
        'Line\n']

    # ctx.io.write and ctx.print
    # need to be guarded for guaranteed
```

(continues on next page)

(continued from previous page)

```
    # write order, or they might get
    # scrambled in with other IO pake is doing

    with ctx.io_lock:
        # Lock, so all these writes come in
        # a defined order when jobs > 1

        for i in data:
           # Add a random short delay in seconds
           # to make things interesting

           time.sleep(random.uniform(0, 0.3))
           ctx.io.write(i)

    # This could get scrambled in the output for
    # the task, because your other sub tasks might
    # be interjecting and printing/writing stuff in
    # between these calls to ctx.print when jobs > 1

    data = ['These', 'Are', 'Somewhere', 'Very', 'Weird']

    for i in data:
           # Add a random short delay in seconds
           # to make things interesting

        time.sleep(random.uniform(0, 0.3))

        ctx.print(i)


@pk.task
def my_task(ctx):
    # Run the sub task 3 times in parallel,
    # passing it the task context

    with ctx.multitask() as mt:
        for i in range(0, 3):
            mt.submit(my_sub_task, ctx)


pake.run(pk, tasks=my_task)
```

Example Output (Will vary of course):

`pake -j 10`

```
===== Executing Task: "my_task"
Hello World, I Come On One Line
Hello World, I Come On One Line
Hello World, I Come On One Line
These
These
Are
Are
These
Somewhere
Very
```

(continues on next page)

(continued from previous page)

```
Are
Somewhere
Somewhere
Weird
Very
Very
Weird
Weird
```

3.8.2 Output synchronization with ctx.call & ctx.subpake

`pake.TaskContext.subpake()`, and `pake.call()` both have an argument named **collect_output** which will do all the work required to synchronize output from sub-pakefiles/processes in a memory efficient manner.

> *Note:*
>
> `pake.subpake()` also has this argument, but you need to pass a lockable context manager object to **collect_output_lock** in order to properly synchronize its output to the **stdout** parameter. `pake.TaskContext.subpake()` does all of this for you and a few extra things to make sure everything works right, so use it for multitasking inside tasks instead. It passes in the `pake.TaskContext.io_lock` object as a lock, just FYI.

When the **collect_output** is **True** and the **silent** parameter of these functions is **False**, they will buffer all process output to a temporary file while the process is doing work.

When the process finishes, theses functions will acquire a lock on `pake.TaskContext.io_lock` and write all their output to the task's IO queue incrementally. This way the sub-pakefile/process output will not get scrambled in with output from other sub tasks that are running concurrently.

Reading process output incrementally from a temporary file after a process completes will occur much faster than it takes for the actual process to finish.

This means that other processes which may have output can do work and write concurrently, and pake only needs to lock the task IO queue when it has to relay the output from a completed process (which is faster than locking while the process is writing).

When pake relays sub-pakefile/process output and **collect_output** is **True**, the output will be read/written in chunks to prevent possible memory issues with processes that produce a lot of output.

The **collect_output** parameter can be bound to a certain value with `functools.partial()`, which works well with `pake.MultitaskContext.map()` and the other methods of the multitasking context.

Example:

```python
import pake

# functools.partial is used for binding argument values to functions

from functools import partial


pk = pake.init()


@pk.task(i=pake.glob('src/*.c'), o=pake.pattern('obj/%.o'))
def compile_c(ctx):

```

(continues on next page)

(continued from previous page)

```
    file_helper = pake.FileHelper(ctx)

    # Make 'obj' directory if it does not exist.
    # This does not complain if it is already there.

    file_helper.makedirs('obj')

    # Generate a command for every invocation of GCC that is needed

    compiler_commands = (['gcc', '-c', i, '-o', o] for i, o in ctx.outdated_pairs)

    # ----

    # Only use collect_output when the number of jobs is greater than 1.

    # Task context functions with collect_output parameters such as
    # ctx.call and ctx.subpake will not degrade back to non-locking
    # behavior on their own when the job count is only 1 and collect_output=True.
    # This is so you can use this feature with a thread or a threadpool you have
    # created yourself if you want to, without pake messing it up automagically.

    # You should turn collect_output off when not running pake in parallel,
    # or when you are not using ctx.call or ctx.subpake from another thread
    # that you have manually created. It will still work if you don't, but it
    # will lock IO and pause the main thread until all process output is collected,
    # even when it does not need be doing that.

    sync_call = partial(ctx.call,
                        collect_output=pk.max_jobs > 1)

    # ^^^ You can bind any other arguments to ctx.call you might need this way too.

    with ctx.multitask() as mt:

        # Apply sync_call to every command
        # in the compiler_commands list with map,
        # and force execution of the returned generator
        # by passing it to a list constructor

        # This will execute GCC in parallel on the main task
        # threadpool if pake's --jobs argument is > 1

        # sync_call will keep GCC's output from becoming
        # scrambled in with other stuff if it happens to
        # print warning information or something

        list(mt.map(sync_call, compiler_args))


pake.run(pk, tasks=compile_c)
```

3.8.3 Sub task exceptions

If an exception occurs inside one of the sub tasks submitted to `pake.MultitaskContext.submit()` or `pake.MultitaskContext.map()`, it will be re-raised in the foreground thread of your pake task at the end of your **with** statement.

The pake task *(your registered task)* will then take over handling of the exception if you do not catch it. It will be wrapped in a `pake.TaskException` which is raised from `pake.Pake.run()` and handled by `pake.run()`.

By default, if more than one task completes with an exception, the one that was submitted first will be the one to have its exception re-raised.

You can set the **aggregate_exceptions** parameter of `pake.TaskContext.multitask()` to **True**, and it will return an executor context that will collect any raised exceptions and add them all to a `pake.AggregateException`. The aggregate exception will then be raised at the end of your **with** statement.

Example:

```python
import pake

pk = pake.init()

class MyException(Exception):
    pass

def my_sub_task():
    raise MyException('Hello World!')

@pk.task
def my_task(ctx):
    with ctx.multitask(aggregate_exceptions=True) as mt:

        # You can also do this, instead of using the parameter
        mt.aggregate_exceptions = True

        for i in range(0, 3):
            mt.submit(my_sub_task)

# Force this example to run with 10 jobs
# regardless of what the command line says

pake.run(pk, tasks=my_task, jobs=10)
```

Output:

```
===== Executing Task: "my_task"

Exception "pake.pake.AggregateException" was raised within task "my_task".

Traceback (most recent call last):
  File "{PAKE_INSTALL_DIR}/pake/pake.py", line 1937, in func_wrapper
    return func(*args, **kwargs)
  File "{PAKEFILE_DIR}/pakefile.py", line 19, in my_task
    mt.submit(my_sub_task)
  File "{PAKE_INSTALL_DIR}/pake/pake.py", line 1228, in __exit__
    self.shutdown()
  File "{PAKE_INSTALL_DIR}/pake/pake.py", line 1225, in shutdown
    raise AggregateException(exceptions)
pake.pake.AggregateException: [MyException('Hello World!',), MyException('Hello World!
↪',), MyException('Hello World!',)]


All Aggregated Exceptions:
```

(continues on next page)

(continued from previous page)

```

Exception Number 1:
===================

Traceback (most recent call last):
  File "{PYTHON_INSTALL_DIR}/lib/concurrent/futures/thread.py", line 55, in run
    result = self.fn(*self.args, **self.kwargs)
  File "{PAKEFILE_DIR}/pakefile.py", line 9, in my_sub_task
    raise MyException('Hello World!')
MyException: Hello World!

Exception Number 2:
===================

Traceback (most recent call last):
  File "{PYTHON_INSTALL_DIR}/lib/concurrent/futures/thread.py", line 55, in run
    result = self.fn(*self.args, **self.kwargs)
  File "{PAKEFILE_DIR}/pakefile.py", line 9, in my_sub_task
    raise MyException('Hello World!')
MyException: Hello World!

Exception Number 3:
===================

Traceback (most recent call last):
  File "{PYTHON_INSTALL_DIR}/lib/concurrent/futures/thread.py", line 55, in run
    result = self.fn(*self.args, **self.kwargs)
  File "{PAKEFILE_DIR}/pakefile.py", line 9, in my_sub_task
    raise MyException('Hello World!')
MyException: Hello World!
```

Aggregate exceptions will be wrapped in a `pake.TaskException` and thrown from `pake.Pake.run()` just like any other exception. `pake.run()` intercepts the task exception and makes sure it gets printed in a way that is readable if it contains an instance of `pake.AggregateException`.

If you are not using a **with** statement, the exception will propagate out of `pake.MultitaskContext.shutdown()` when you call it manually, unless you pass **wait=False**, in which case no exceptions will be re-raised.

3.9 Manipulating Files / Dirs With pake.FileHelper

`pake.FileHelper` contains several useful filesystem manipulation methods that are common in software builds. Operations include creating full directory trees, glob removal of files and directories, file touch etc..

The `pake.FileHelper` class takes a single optional argument named **printer**.

The passed object should implement a **print(*args)** function.

If you pass it a `pake.TaskContext` instance from your tasks single argument, it will print information about file system operations to the tasks IO queue as they are being performed.

Each method can turn off this printing by using a **silent** option argument that is common to all class methods.

If you construct `pake.FileHelper` without an argument, all operations will occur silently.

3.9.1 File / Folder creation methods

```
@pk.task
def my_build(ctx):

    fh = pake.FileHelper(ctx)

    # Create a directory or an entire directory tree

    fh.makedirs('dist/bin')

    # Touch a file

    fh.touch('somefile.txt')
```

Output:

```
===== Executing Task: "my_build"
Created Directory(s): "dist/bin"
Touched File: "somefile.txt"
```

3.9.2 Copy / Move methods

```
@pk.task
def my_build(ctx):

fh = pake.FileHelper(ctx)

# Recursively copy and entire directory tree.
# In this case, 'bin' will be copied into 'dist'
# as a subfolder.

fh.copytree('bin', 'dist/bin')


# Recursively move an entire directory tree
# and its contents.  In this case, 'lib' will
# be moved into 'dist' as a subfolder.

fh.move('lib', 'dist/lib')


# Copy a file to a directory without
# renaming it.

fh.copy('LICENCE.txt', 'dist')

# Copy with rename

fh.copy('LICENCE.txt', 'dist/licence.txt')


# Move a file to a directory without
# renaming it.
```

(continues on next page)

(continued from previous page)

```
fh.move('README.txt', 'dist')

# Move with rename

fh.move('README.rtf', 'dist/readme.rtf')
```

Output:

```
===== Executing Task: "my_build"
Copied Tree: "bin" -> "dist/bin"
Moved Tree: "lib" -> "dist/lib"
Copied File: "LICENCE.txt" -> "dist"
Copied File: "LICENCE.txt" -> "dist/licence.txt"
Moved File: "README.txt" -> "dist"
Moved File: "README.rtf" -> "dist/readme.rtf"
```

3.9.3 Removal / Clean related methods

```
@pk.task
def my_clean(ctx):

   fh = pake.FileHelper(ctx)


   # Glob delete all files under the 'obj' directory

   fh.glob_remove('obj/*.o')


   # Delete all sub directories of 'stuff'

   fh.glob_remove_dirs('stuff/*')


   # Remove a directory tree, does nothing if 'build_dir'
   # does not exist.  Unless the must_exist argument is
   # set to True.

   fh.rmtree('build_dir')


   # Remove a file, does nothing if 'main.exe' does not
   # exist.  Unless the must_exist argument is set to True

   fh.remove('main.exe')
```

Output:

```
===== Executing Task: "my_clean"
Glob Removed Files: "obj/*.o"
Glob Removed Directories: "stuff/*"
Removed Directory(s): "build_dir"
Removed File: "main.exe"
```

3.10 Running Commands / Sub Processes

3.10.1 TaskContext.call

The `pake.TaskContext` object passed into each task contains methods for calling sub-processes in a way that produces user friendly error messages and halts the execution of pake if an error is reported by the given process.

`pake.TaskContext.call()` can be used to run a program and direct all of its output (stdout and stderr) to the tasks IO queue.

It will raise a `pake.TaskSubprocessException` on non-zero return codes by default unless you specify **ignore_errors=True**.

If you specify for **call** to ignore errors, it will always return the process's return code regardless of whether it was non-zero or not.

It can be used to test return codes like `pake.TaskContext.check_call()`, but it is preferable to use the later method for that purpose since `pake.TaskContext.call()` prints to the task IO queue by default. (unless you specify **silent=True** and **print_cmd=False**).

`pake.TaskContext.call()` is designed primarily for handling large amounts of process output and reporting it back when an error occurs without crashing pake, which is accomplished by duplicating the process output to a temporary file on disk and later reading it back incrementally when needed.

Examples:

```python
# 'call' can have its arguments passed in several different ways

@pk.task(o='somefile.txt')
def my_task(ctx):

    # Command line passed as a list..
    ctx.call(['echo', 'Hello!'])

    # Iterables such as the outputs property of the context
    # will be flattened.  String objects are not considered
    # for flattening which allows this sort of syntax

    ctx.call(['touch', ctx.outputs]) # We know there is only one output


    # The above will also work when using varargs

    ctx.call('touch', ctx.outputs)


    # Command line passed as a string

    ctx.call('echo "goodbye!"')

    # Try some command and ignore any errors (non-zero return codes)
    # Otherwise, 'call' raises a 'pake.TaskSubprocessException' on non-zero
    # return codes.

    ctx.call(['do_something_bad'], ignore_errors=True)


# A realistic example for compiling objects from C
```

(continues on next page)

(continued from previous page)

```
@pk.task(i=pake.glob('src/*.c'), o=pake.pattern('obj/%.o'))
def compile_c(ctx):
    for i, o in ctx.outdated_pairs:
        ctx.call('gcc', '-c', i, '-o', o)


# And with multitasking, the simple way

@pk.task(i=pake.glob('src/*.c'), o=pake.pattern('obj/%.o'))
def compile_c(ctx):
    with ctx.multitask() as mt:
        for i, o in ctx.outdated_pairs:
            mt.submit(ctx.call, ['gcc', '-c', i, '-o', o])


# With multitasking, the fancy way

@pk.task(i=pake.glob('src/*.c'), o=pake.pattern('obj/%.o'))
def compile_c(ctx):
    with ctx.multitask() as mt:

        # Force enumeration over the returned generator by constructing a temporary␣
↪list..
        # the 'ctx.map' function yields 'Future' instances

        list(ctx.map(ctx.call, (['gcc', '-c', i, '-o', o] for i, o in ctx.outdated_
↪pairs)))
```

3.10.2 TaskContext.check_output

pake.TaskContext.check_output() can be used to read all the output from a command into a bytes object. The args parameter of **check_output** and in general all functions dealing with calling system commands allow for identical syntax, including nested lists and such.

The reasoning or using this over the built in `subprocess.check_output()` is that if an error occurs in the subprocess, pake will be able to print more comprehensible error information to the task output.

pake.TaskContext.check_output() differs from `subprocess.check_output()` in that you cannot specify an **stderr** parameter, and an **ignore_errors** option is added which can prevent the method from raising an exception on non zero return codes from the process. All of the process's **stderr** is directed to its **stdout**.

ignore_errors allows you to directly return the output of a command even if it errors without having to handle an exception to get the output.

pake.TaskContext.check_output() returns a **bytes** object, which means you need to call **decode** on it if you want the output as a string.

Examples:

```
# 'which' is a unix command that returns the full path of a command's binary.
# The exit code is non-zero if the command given does not exist.  So
# it will be easy enough to use for this example.

@pk.task
def my_task(ctx):
```

(continues on next page)

(continued from previous page)

```
    # Print the full path of the default C compiler on linux

    ctx.print(ctx.check_output('which', 'cc').decode())

    # Check if some command exists

    if ctx.check_output(['which', 'some_command'],
                        ignore_errors=True).decode().strip() != '':

        ctx.print('some_command exists')

    # Using an exception handler

    try:
        path = ctx.check_output('which', 'gcc').decode()
        ctx.print('gcc exists!, path:', path)
    except pake.TaskSubprocessException:
        pass
```

3.10.3 TaskContext.check_call

pake.TaskContext.check_call() has an identical signature to *pake.TaskContext.check_output()*, except it returns the return code of the called process.

The **ignore_errors** argument allows you to return the value of non-zero return codes without having to handle an exception such as with `subprocess.check_call()` from pythons built in subprocess module.

In addition, if an exception is thrown, pake will be able to print comprehensible error info about the location of the exception to the task IO queue while avoiding a huge stack trace. The same is true for the other functions dealing with processes in the task context.

Examples:

```
# using the 'which' command here again for this example...

@pk.task
def my_task(ctx):

    # Check if some command exists, a better way on linux at least

    if ctx.check_call(['which', 'some_command'],
                      ignore_errors=True) == 0:

        ctx.print('some_command exists')

    # Using an exception handler

    try:
        ctx.check_call('which', 'gcc')
        ctx.print('gcc exists!')
    except pake.TaskSubprocessException:
        pass
```

3.10.4 pake.process module methods

The *pake.process* module provides thin wrappers around the built in python `subprocess` module methods.

Primarily: `subprocess.call()`, `subprocess.check_call()` and `subprocess.check_output()`.

The corresponding wrappers are: *pake.process.call()*, *pake.process.check_call()* and *pake.process.check_output()*.

The wrappers exist mostly to allow calling sub-processes with a similar syntax to `pake.Pake.call()` and friends.

IE. They can be called with variadic arguments, and will also flatten any non string iterables passed to the ***args** parameter.

Example:

```python
import sys
from pake import process


def run_python_silent(*args):
    # sys.executable and *args go into the variadic argument, the
    # *args iterable is flattened out for you

    # Returns the return code
    return process.call(sys.executable, args,
                        stdout=process.DEVNULL,
                        stderr=process.DEVNULL)
```

They also raise exceptions similar to those from the `subprocess` module, however the exceptions behave nicer if they occur inside of a task.

See: *pake.process.TimeoutExpired* and *pake.process.CalledProcessException*.

Which are analogs for `subprocess.TimeoutExpired` and `subprocess.CalledProcessException`.

3.11 Running Sub Pakefiles

Pake is able to run itself through the use of *pake.TaskContext.subpake()* and *pake.subpake()*.

pake.subpake() is meant to be used outside of tasks, and can even be called before pake is initialized.

pake.TaskContext.subpake() is preferred for use inside of tasks because it handles writing to the task's output queue for you, without having to specify extra parameters to *pake.subpake()* to get it working correctly.

A *pake.TaskContext* instance is passed into the single argument of each task function, which you can in turn call **subpake** from.

Defines can be exported to pakefiles ran with the **subpake** functions using *pake.export()*.

pake.subpake() and *pake.TaskContext.subpake()* use the **–stdin-defines** option of pake to pass exported define values into the new process instance, which means you can overwrite your exported define values with **-D/–define** in the subpake command arguments if you need to.

Export / Subpake Example:

```python
import pake

pk = pake.init()
```

(continues on next page)

(continued from previous page)

```python
# Try to get the CC define from the command line,
# default to 'gcc'.

CC = pk.get_define('CC', 'gcc')

# Export the CC variable's value to all invocations
# of pake.subpake or ctx.subpake as a define that can be
# retrieved with pk.get_define()

pake.export('CC', CC)


# You can also export lists, dictionaries sets and tuples,
# as long as they only contain literal values.
# Literal values being: strings, integers, floats; and
# other lists, dicts, sets and tuples.  Collections must only
# contain literals, or objects that repr() into a parsable literal.

pake.export('CC_FLAGS', ['-Wextra', '-Wall'])


# Nesting works with composite literals,
# as long as everything is a pure literal or something
# that str()'s into a literal.

pake.export('STUFF',
            ['you',
             ['might',
              ('be',
               ['a',
                {'bad' :
                     ['person', ['if', {'you', 'do'}, ('this',) ]]
                 }])])])


# This export will be overrode in the next call
pake.export('OVERRIDE_ME', False)


# Execute outside of a task, by default the stdout/stderr
# of the subscript goes to this scripts stdout.  The file
# object to which stdout gets written to can be specified
# with pake.subpake(..., stdout=(file))

# This command also demonstrates that you can override
# your exports using the -D/--define option

pake.subpake('sometasks/pakefile.py', 'dotasks', '-D', 'OVERRIDE_ME=True')


# This task does not depend on anything or have any inputs/outputs
# it will basically only run if you explicitly specify it as a default
# task in pake.run, or specify it on the command line

@pk.task
def my_phony_task(ctx):
    # Arguments are passed in a variadic parameter...
```

(continues on next page)

(continued from previous page)

```
    # Specify that the "foo" task is to be ran.
    # The scripts output is written to this tasks output queue

    ctx.subpake('library/pakefile.py', 'foo')



# Run this pake script, with a default task of 'my_phony_task'

pake.run(pk, tasks=my_phony_task)
```

Output from the example above:

```
*** enter subpake[1]:
pake[1]: Entering Directory "(REST OF PATH...)/paketest/sometasks"
===== Executing Task: "dotasks"
Do Tasks
pake[1]: Exiting Directory "(REST OF PATH...)/paketest/sometasks"
*** exit subpake[1]:
===== Executing Task: "my_phony_task"
*** enter subpake[1]:
pake[1]: Entering Directory "(REST OF PATH...)/paketest/library"
===== Executing Task: "foo"
Foo!
pake[1]: Exiting Directory "(REST OF PATH...)/paketest/library"
*** exit subpake[1]:
```

CHAPTER 4

Module Index

- modindex

Python Module Index

Index